Reference. The Processing Language was designed to facilitate the creation of sophisticated visual structures. Made by Chrisir on processing forum

http://www.processing.org/reference/

INDEX

Name

() (parentheses)

Examples

```
int a;
a = (4 + 3) * 2;       // Grouping expressions
if (a > 10) {          // Containing expressions
  line(a, 0, a, 100);  // Containing a list of parameters
}
```

Description Grouping and containing expressions and parameters. Parentheses have multiple functions relating to functions and structures. They are used to contain a list of parameters passed to functions and control structures and they are used to group expressions to control the order of execution. Some functions have no parameters and in this case, the space between parentheses is blank.

Syntax

```
function()
function(p1, ..., pN)
structure(expression)
```

Parameters

function	any function
p1, ..., pN	list of parameters specific to the function
structure	Control structure such as if, for, while
expressions	any valid expression or group of expression

Related , (comma)

Name

, (comma)

Examples

```
// Comma used to separate a list of variable declarations
int a=20, b=30, c=80;

// Comma used to separate a list of values assigned to an array
int[] d = { 20, 60, 80 };

// Comma used to separate a list of parameters passed to a
function
line(a, b, c, b);
line(d[0], d[1], d[2], d[1]);
```

Description Separates parameters in function calls and elements during assignment.

Syntax

```
value1, ..., valueN
```

Parameters **value1, ..., valueN** any int, float, byte, boolean, color, char, String

---

Name

. (dot)

Examples

```
// Declare and construct two objects (h1 and h2) of the class
HLine
```

```
HLine h1 = new HLine(20, 1.0);
HLine h2 = new HLine(50, 5.0);

void setup() {
  size(200, 200);
}

void draw() {
  if (h2.speed > 1.0) {  // Dot syntax can be used to get a value
    h2.speed -= 0.01;    // or set a value.
  }
  h1.update();  // Calls the h1 object's update() function
  h2.update();  // Calls the h2 object's update() function
}

class HLine {  // Class definition
  float ypos, speed;  // Data
  HLine (float y, float s) {  // Object constructor
    ypos = y;
    speed = s;
  }
  void update() {  // Update method
    ypos += speed;
    if (ypos > width) {
      ypos = 0;
    }
    line(0, ypos, width, ypos);
  }
}
```

Description Provides access to an object's methods and data. An object is one instance of a class and may contain both methods (object functions) and data (object variables and constants), as specified in the class definition. The dot operator directs the program to the information encapsulated within an object.

Syntax
```
object.method()
object.data
```

Parameters
object the object to be accessed
method() a method encapsulated in the object
data a variable or constant encapsulated in the object

Related Object

Name

/* */ (multiline comment)

Examples
```
/*
   Draws two lines which divides the window
   into four quadrants. First draws a horizontal
   line and then the vertical line
*/
line(0, 50, 100, 50);
line(50, 0, 50, 100);
```

Description Explanatory notes embedded within the code. Comments are used to remind yourself and to inform others about the function of your program. Multiline comments are used for large text descriptions of code or to comment out chunks of code while debugging applications. Comments are ignored by the compiler

Syntax
```
/*
  comment
*/
```

Parameters **comment** any sequence of characters

Related // (comment)
/** */ (doc comment)

Name

/** */ (doc comment)

Examples

```
/**
   Draws two lines which divides the window
   into four quadrants. First draws a horizontal
   line and then the vertical line
*/
line(0, 50, 100, 50);
line(50, 0, 50, 100);
```

Description Explanatory notes embedded within the code and written to the "index.html" file created when the code is exported. Doc comments (documentation comments) are used for sharing a description of your sketch when the program is exported. Export the code by hitting the "Export" button on the Toolbar.

Syntax

```
/**
  comment
*/
```

Parameters **comment** any sequence of characters

Related // (comment)
/* */ (multiline comment)

Name

// (comment)

Examples

```
// Draws two lines which divides the window
// into four quadrants
line(0, 50, 100, 50);  // Draw the horizontal line
line(50, 0, 50, 100);  // Draw the vertical line
```

Description Explanatory notes embedded within the code. Comments are used to remind yourself and to inform others about the details of the code. Single-line comments are signified with the two forward slash characters. Comments are ignored by the compiler.

Syntax

```
// comment
```

Parameters **comment** any sequence of characters

Related /* */ (multiline comment)
/** */ (doc comment)

Name

; (semicolon)

Examples

```
int a;       // Declaration statement
a = 30;      // Assignment statement
println(i);  // Function statement
```

Description A statement terminator which separates elements of the program. A statement is a complete instruction to the computer and the semicolon is used to separate instructions (this is similar to the period "." in written English). Semicolons are also used to separate the different elements of a **for** structure.

Syntax
```
statement;
```
Parameters **statement** a single statement to execute

Related for

Name

= (assign)

Examples
```
int a;
a = 30;  // Assigns for value 30 to the variable 'a'
a = a + 40;  // Assigns the value 70 to the variable 'a'
```
Description Assigns a value to a variable. The "=" sign does not mean "equals", but is used to place data within a variable. The "=" operator is formally called the assignment operator. There are many different types of variables (int, floats, strings, etc.) and the assignment operator can only assign values which are the same type as the variable it is assigning. For example, if the variable is of type **int**, the value must also be an **int**.

Syntax
```
var = value
```
Parameters **var** any valid variable name

value any value of the same type as the variable. For example, if the variable is of type "int", the value must also be an int

Related
+= (add assign)
-= (subtract assign)

Name

[] (array access)

Examples
```
int[] numbers = new int[3];
numbers[0] = 90;
numbers[1] = 150;
numbers[2] = 30;
int a = numbers[0] + numbers[1]; // Sets variable 'a' to 240
int b = numbers[1] + numbers[2]; // Sets variable 'b' to 180
```
Description The array access operator is used to specify a location within an array. The data at this location can be defined with the syntax **array[element] = value** and read with the syntax **value = array[element]** as shown in the above example.

Syntax
```
datatype[]
array[element]
```
Parameters **datatype** any primitive or compound datatype, including user-defined classes

array any valid variable name

element int: must not exceed the length of the array minus 1

Related Array

Name

{} (curly braces)

Examples
```
int[] a = { 5, 20, 25, 45, 70 };

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

void draw() {
```

```
  for (int i=0; i < a.length; i++) {
    line(0, a[i], 50, a[i]);
  }
}
```

Description Define the beginning and end of functions blocks and statement blocks such as the **for** and **if** structures. Curly braces are also used for defining inital values in array declarations.

Syntax
```
{ statements }
{ ele0, ..., eleN }
```

Parameters
statements any sequence of valid statements
ele0, ..., eleN list of elements separated by commas

Related () (parentheses)

Name

catch

Examples
```
BufferedReader reader;
String line;

void setup() {
  // Open the file from the createWriter() example
  reader = createReader("positions.txt");
}

void draw() {
  try {
    line = reader.readLine();
  } catch (IOException e) {
    e.printStackTrace();
    line = null;
  }
  if (line == null) {
    // Stop reading because of an error or file is empty
    noLoop();
  } else {
    String[] pieces = split(line, TAB);
    int x = int(pieces[0]);
    int y = int(pieces[1]);
    point(x, y);
  }
}
```

Description The **catch** keyword is used with **try** to handle exceptions. Sun's Java documentation defines an exception as "an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions." This could be, for example, an error while a file is read.

Syntax
```
try {
  tryStatements
} catch (exception) {
  catchStatements
}
```

Parameters
tryStatements if this code throws an exception, then the code in "catch" is run
exception the Java exception that was thrown
catchStatements code that handles the exception

Name

class

```
// Declare and construct two objects (h1, h2) from the class
HLine
HLine h1 = new HLine(20, 2.0);
HLine h2 = new HLine(50, 2.5);

void setup()
{
  size(200, 200);
  frameRate(30);
}

void draw() {
  background(204);
  h1.update();
  h2.update();
}
```

Examples

```
class HLine {
  float ypos, speed;
  HLine (float y, float s) {
    ypos = y;
    speed = s;
  }
  void update() {
    ypos += speed;
    if (ypos > height) {
      ypos = 0;
    }
    line(0, ypos, width, ypos);
  }
}
```

Description Keyword used to indicate the declaration of a class. A class is a composite of fields (data) and methods (functions that are a part of the class) which may be instantiated as objects. The first letter of a class name is usually uppercase to separate it from other kinds of variables. A related tutorial on Object-Oriented Programming is hosted on the Oracle website.

Syntax

```
class ClassName {
  statements
}
```

Parameters **ClassName** Any valid variable name
statements any valid statements

Related Object

Name

draw()

Examples

```
float yPos = 0.0;

void setup() {  // setup() runs once
  size(200, 200);
  frameRate(30);
}

void draw() {  // draw() loops forever, until stopped
  background(204);
  yPos = yPos - 1.0;
  if (yPos < 0) {
```

```
    yPos = height;
  }
  line(0, yPos, width, yPos);
}
```

```
void setup() {
  size(200, 200);
}

// Although empty here, draw() is needed so
// the sketch can process user input events
// (mouse presses in this case).
void draw() { }

void mousePressed() {
  line(mouseX, 10, mouseX, 90);
}
```

Description

Called directly after **setup()**, the **draw()** function continuously executes the lines of code contained inside its block until the program is stopped or **noLoop()** is called. **draw()** is called automatically and should never be called explicitly.

It should always be controlled with **noLoop()**, **redraw()** and **loop()**. After **noLoop()** stops the code in **draw()** from executing, **redraw()** causes the code inside **draw()** to execute once, and **loop()** will cause the code inside **draw()** to resume executing continuously.

The number of times **draw()** executes in each second may be controlled with the **frameRate()** function.

There can only be one **draw()** function for each sketch, and **draw()** must exist if you want the code to run continuously, or to process events such as **mousePressed()**. Sometimes, you might have an empty call to **draw()** in your program, as shown in the above example.

Syntax

```
draw()
```

Returns

void

Related

setup()
loop()
noLoop()
redraw()
frameRate

Name

exit()

Examples

```
void draw() {
  line(mouseX, mouseY, 50, 50);
}

void mousePressed() {
  exit();
}
```

Description

Quits/stops/exits the program. Programs without a **draw()** function exit automatically after the last line has run, but programs with **draw()** run continuously until the program is manually stopped or **exit()** is run.

Rather than terminating immediately, **exit()** will cause the sketch to exit after **draw()** has completed (or after **setup()** completes if called during the **setup()** function).

For Java programmers, this is *not* the same as System.exit(). Further, System.exit() should not be used because closing out an application while **draw()** is running may cause a crash (particularly with P3D).

Syntax `exit()`

Returns void

Name

extends

Examples

```
DrawDot dd1 = new DrawDot(50, 80);

void setup() {
  size(200, 200);
}

void draw() {
  dd1.display();
}

class Dot {
  int xpos, ypos;
}

class DrawDot extends Dot {
  DrawDot(int x, int y) {
    xpos = x;
    ypos = y;
  }
  void display() {
    ellipse(xpos, ypos, 200, 200);
  }
}
```

Description Allows a new class to *inherit* the methods and data fields (variables and constants) from an existing class. In code, state the name of the new class, followed by the keyword **extends** and the name of the *base class*. The concept of inheritance is one of the fundamental principles of object oriented programming.

Note that in Java, and therefore also Processing, you cannot extend a class more than once. Instead, see **implements**.

Related
class
super
implements

Name

false

Examples

```
rect(30, 20, 50, 50);
boolean b = false;
if (b == false) {
```

```
  line(20, 10, 90, 80);  // This line is drawn
} else {
  line(20, 80, 90, 10);  // This line is not drawn
}
```

Description Reserved word representing the logical value "false". Only variables of type **boolean** may be assigned the value **false**.

Related true
boolean

Name

final

Examples

```
final float constant = 12.84753;
println(constant);  // Prints "12.84753" to the console
constant += 12.84;  // ERROR! It's not possible to change a
"final" value
```

Description Keyword used to state that a value, class, or method can't be changed. If the **final** keyword is used to define a variable, the variable can't be changed within the program. When used to define a class, a **final** class cannot be subclassed. When used to define a function or method, a **final** method can't be overridden by subclasses.

This keyword is an essential part of Java programming and is not usually used with Processing. Consult a Java language reference or tutorial for more information.

Name

implements

Examples

```
// No example here yet
```

Description Implements an *interface* or group of *interfaces*. Interfaces are used to establish a protocol between classes; they establish the form for a class (method names, return types, etc.) but no implementation. After implementation, an interface can be used and extended like any other class.

Because Java doesn't allow extending more than one class at a time, you can create interfaces instead, so specific methods and fields can be found in the class which implements it. A Thread is an example; it implements the "Runnable" interface, which means the class has a method called "public void run()" inside it.

Related extends

Name

import

Examples

```
import processing.pdf.*;

void setup() {
  size(screenWidth, screenHeight, PDF);
}

void draw() {
```

```
  line(0, 0, width, height);
  line(0, height, width, 0);
}
```

Description The keyword **import** is used to load a library into a Processing sketch. A library is one or more classes that are grouped together to extend the capabilities of Processing. The * character is often used at the end of the import line (see the code example above) to load all of the related classes at once, without having to reference them individually.

The import statement will be created for you by selecting a library from the "Import Library..." item in the Sketch menu.

Syntax `import libraryName`

Parameters **libraryName** the name of the library to load (e.g. "processing.pdf.*")

Name

loop()

Examples

```
void setup() {
  size(200, 200);
  noLoop();  // draw() will not loop
}

float x = 0;

void draw() {
  background(204);
  x = x + .1;
  if (x > width) {
    x = 0;
  }
  line(x, 0, x, height);
}

void mousePressed() {
  loop();  // Holding down the mouse activates looping
}

void mouseReleased() {
  noLoop();  // Releasing the mouse stops looping draw()
}
```

Description By default, Processing loops through **draw()** continuously, executing the code within it. However, the **draw()** loop may be stopped by calling **noLoop()**. In that case, the **draw()** loop can be resumed with **loop()**.

Syntax `loop()`

Returns void

Related noLoop()
redraw()
draw()

Name

new

Examples

```
HLine h1 = new HLine();
float[] speeds = new float[3];
```

```
float ypos;

void setup() {
  size(200, 200);
  speeds[0] = 0.1;
  speeds[1] = 2.0;
  speeds[2] = 0.5;
}

void draw() {
  ypos += speeds[int(random(3))];
  if (ypos > width) {
    ypos = 0;
  }
  h1.update(ypos);
}

class HLine {
  void update(float y) {
    line(0, y, width, y);
  }
}
```

Description Creates a "new" object. The keyword **new** is typically used similarly to the applications in the above example. In this example, a new object "h1" of the datatype "HLine" is created. On the following line, a new array of floats called "speeds" is created.

Name

noLoop()

Examples

```
void setup() {
  size(200, 200);
  noLoop();
}

void draw() {
  line(10, 10, 190, 190);
}
```

```
void setup() {
  size(200, 200);
}

float x = 0.0;

void draw() {
  background(204);
  x = x + 0.1;
  if (x > width) {
    x = 0;
  }
  line(x, 0, x, height);
}

void mousePressed() {
  noLoop();
}
```

```
void mouseReleased() {
  loop();
}
```

```
boolean someMode = false;

void setup() {
  noLoop();
}

void draw() {
  if (someMode) {
    // do something
  }
}

void mousePressed() {
  someMode = true;
  redraw();  // or loop()
}
```

Description

Stops Processing from continuously executing the code within **draw()**. If **loop()** is called, the code in **draw()** begins to run continuously again. If using **noLoop()** in **setup()**, it should be the last line inside the block.

When **noLoop()** is used, it's not possible to manipulate or access the screen inside event handling functions such as **mousePressed()** or **keyPressed()**. Instead, use those functions to call **redraw()** or **loop()**, which will run **draw()**, which can update the screen properly. This means that when **noLoop()** has been called, no drawing can happen, and functions like **saveFrame()** or **loadPixels()** may not be used.

Note that if the sketch is resized, **redraw()** will be called to update the sketch, even after **noLoop()** has been specified. Otherwise, the sketch would enter an odd state until **loop()** was called.

Syntax `noLoop()`

Returns void

Related loop()
redraw()
draw()

Name

null

Examples

```
String content = "It is a beautiful day.";
String[] results;  // Declare empty String array

results = match(content, "orange");
// The match statement above will fail to find
// the word "orange" in the String 'content', so
// it will return a null value to 'results'.

if (results == null) {
  println("Value of 'results' is null.");  // This line is
printed
} else {
  println("Value of 'results' is not null!");  // This line is
```

```
not printed
}
```

Description Special value used to signify the target is not a valid data element. In Processing, you may run across the keyword **null** when trying to access data which is not there.

Name

popStyle()

Examples

```
ellipse(0, 50, 33, 33);  // Left circle

pushStyle();  // Start a new style
strokeWeight(10);
fill(204, 153, 0);
ellipse(50, 50, 33, 33);  // Middle circle
popStyle();  // Restore original style

ellipse(100, 50, 33, 33);  // Right circle
```

```
ellipse(0, 50, 33, 33);  // Left circle

pushStyle();  // Start a new style
strokeWeight(10);
fill(204, 153, 0);
ellipse(33, 50, 33, 33);  // Left-middle circle

pushStyle();  // Start another new style
stroke(0, 102, 153);
ellipse(66, 50, 33, 33);  // Right-middle circle
popStyle();  // Restore the previous style

popStyle();  // Restore original style

ellipse(100, 50, 33, 33);  // Right circle
```

Description The **pushStyle()** function saves the current style settings and **popStyle()** restores the prior settings; these functions are always used together. They allow you to change the style settings and later return to what you had. When a new style is started with **pushStyle()**, it builds on the current style information. The **pushStyle()** and **popStyle()** functions can be embedded to provide more control (see the second example above for a demonstration.)

Syntax `popStyle()`

Returns void

Related pushStyle()

Name

private

Description Keyword used to disallow other classes access the fields and methods within a class. The **private** keyword is used before a field or method that you want to be available only within the class. In Processing, all fields and methods are public unless otherwise specified by the **private** keyword.

This keyword is an essential part of Java programming and is not usually used with Processing. Consult a Java language reference or tutorial for more information.

Related public

Name

public

Examples

Description

Keyword used to provide other classes access the fields and methods within a class. The **public** keyword is used before a field or method that you want to make available. In Processing, all fields and methods are public unless otherwise specified by the **private** keyword.

This keyword is an essential part of Java programming and is not usually used with Processing. Consult a Java language reference or tutorial for more information.

Related private

Name

pushStyle()

```
ellipse(0, 50, 33, 33);  // Left circle

pushStyle();  // Start a new style
strokeWeight(10);
fill(204, 153, 0);
ellipse(50, 50, 33, 33);  // Middle circle
popStyle();  // Restore original style

ellipse(100, 50, 33, 33);  // Right circle
```

Examples

```
ellipse(0, 50, 33, 33);  // Left circle

pushStyle();  // Start a new style
strokeWeight(10);
fill(204, 153, 0);
ellipse(33, 50, 33, 33);  // Left-middle circle

pushStyle();  // Start another new style
stroke(0, 102, 153);
ellipse(66, 50, 33, 33);  // Right-middle circle
popStyle();  // Restore previous style

popStyle();  // Restore original style

ellipse(100, 50, 33, 33);  // Right circle
```

Description

The **pushStyle()** function saves the current style settings and **popStyle()** restores the prior settings. Note that these functions are always used together. They allow you to change the style settings and later return to what you had. When a new

style is started with **pushStyle()**, it builds on the current style information. The **pushStyle()** and **popStyle()** functions can be embedded to provide more control. (See the second example above for a demonstration.)

The style information controlled by the following functions are included in the style: fill(), stroke(), tint(), strokeWeight(), strokeCap(), strokeJoin(), imageMode(), rectMode(), ellipseMode(), shapeMode(), colorMode(), textAlign(), textFont(), textMode(), textSize(), textLeading(), emissive(), specular(), shininess(), ambient()

Syntax `pushStyle()`

Returns void

Related popStyle()

Name

redraw()

Examples

```
float x = 0;

void setup() {
  size(200, 200);
  noLoop();
}

void draw() {
  background(204);
  line(x, 0, x, height);
}

void mousePressed() {
  x += 1;
  redraw();
}
```

Description Executes the code within **draw()** one time. This functions allows the program to update the display window only when necessary, for example when an event registered by **mousePressed()** or **keyPressed()** occurs.

In structuring a program, it only makes sense to call redraw() within events such as **mousePressed()**. This is because **redraw()** does not run **draw()** immediately (it only sets a flag that indicates an update is needed).

The **redraw()** function does not work properly when called inside **draw()**. To enable/disable animations, use **loop()** and **noLoop()**.

Syntax `redraw()`

Returns void

Related draw()
loop()
noLoop()
frameRate

Name

return

Examples

```
int val = 30;

void draw() {
  int t = timestwo(val);
  println(t);
}

// The first 'int' in the function declaration
// specifies the type of data to be returned.
int timestwo(int dVal) {
  dVal = dVal * 2;
  return dVal;  // Returns an int of 60, in this case
}
```

```
int[] vals = {10, 20, 30};

void draw() {
  int[] t = timestwo(vals);
  println(t);
  noLoop();
}

int[] timestwo(int[] dVals) {
  for (int i = 0; i < dVals.length; i++) {
    dVals[i] = dVals[i] * 2;
  }
  return dVals;  // Returns an array of 3 ints: 20, 40, 60
}
```

```
void draw() {
  background(204);
  line(0, 0, width, height);
  if (mousePressed) {
    return;  // Break out of draw(), skipping the line statement
below
  }
  line(0, height, width, 0);  // Executed only if mouse is not
pressed
}
```

Description

Keyword used to indicate the value to return from a function. The value being returned must be the same datatype as defined in the function declaration. Functions declared with **void** can't return values and shouldn't include a return value.

The keyword **return** may also be used to break out of a function, thus not allowing the program to the remaining statements. (See the third example above.)

Syntax

```
type function {
  statements
  return value
}
```

Parameters

type	boolean, byte, char, int, float, String, boolean[], byte[], char[], int[], float[], or String[]
function	the function that is being defined
statements	any valid statements
value	must be the same datatype as the "type" parameter

Name

setup()

Examples

```
void setup() {
  size(200, 200);
  background(0);
  noStroke();
  fill(102);
}
```

```
int a = 0;

void draw() {
  rect(a++%width, 10, 2, 80);
}
```

Description The **setup()** function is called once when the program starts. It's used to define initial enviroment properties such as screen size and background color and to load media such as images and fonts as the program starts. There can only be one **setup()** function for each program and it shouldn't be called again after its initial execution. Note: Variables declared within **setup()** are not accessible within other functions, including **draw()**.

Syntax `setup()`

Returns void

Related size()
loop()
noLoop()
draw()

Name

static

Examples

```
void setup() {
  MiniClass mc1 = new MiniClass();
  MiniClass mc2 = new MiniClass();
  println( mc1.y );   // Prints "10" to the console
  MiniClass.y += 10;  // The 'y' variable is shared by 'mc1' and
'mc2'
  println( mc1.y );   // Prints "20" to the console
  println( mc2.y );   // Prints "20" to the console
}

static class MiniClass {
  static int y = 10;  // Class variable
}
```

```
void setup() {
  println(MiniClass.add(3, 4));  // Prints "7" to the console
}

static class MiniClass {
  static int add(int x, int y) {
    return(x + y);
  }
}
```

Description Keyword used to define a variable as a "class variable" and a method as a "class

method." When a variable is declared with the **static** keyword, all instances of that class share the same variable. When a class is defined with the **static** keyword, it's methods can be used without making an instance of the class. The above examples demonstrate each of these uses.

This keyword is an essential part of Java programming and is not usually used with Processing. Consult a Java language reference or tutorial for more information.

Name

super

Examples

```
// This example is a code fragment;
// it will not compile on its own.

// Create the DragDrop subclass from
// the Button class. Button becomes
// the superclass of DragDrop.
class DragDrop extends Button {
  int xoff, yoff;
  DragDrop(int x, int y) {
    // Runs the superclass' constructor
    super(x, y);
  }
  void press(int mx, int my) {
    // Runs the superclass' press() method
    super.press();
    xoff = mx;
    yoff = my;
  }
}
```

Description Keyword used to reference the superclass of a subclass.

Related class
extends

Name

this

Examples

```
float ypos = 50;

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

void draw() {
  line(0, 0, 100, ypos);
  // "this" references the Processing sketch,
  // and is not necessary in this case
  this.ypos = 100;
  line(0, 0, 100, ypos);
}
```

```
import processing.video.*;
```

```
Movie myMovie;

void setup() {
  size(200, 200);
  background(0);
  // "this" references the Processing sketch
  myMovie = new Movie(this, "totoro.mov");
  myMovie.loop();
}

void draw() {
  if (myMovie.available()) {
    myMovie.read();
  }
  image(myMovie, 0, 0);
}
```

Description Refers to the current object (i.e., "this object"), which will change depending on the context in which **this** is referenced. In Processing, it's most common to use **this** to pass a reference from the current object into one of the libraries.

The keyword **this** can also be used to reference an object's own method from within itself, but such usage is typically not necessary. For example, if you are calling the **filter()** method of a **PImage** object named **tree** from another object, you would write **tree.filter()**. To call this method inside the PImage object itself, one could simply write **filter()** or, more explicitly, **this.filter()**. The additional level of specificity in **this.filter()** is not necessary, as it is always implied.

Name

true

Examples

```
rect(30, 20, 50, 50);
boolean b = true;
if (b == true) {
  line(20, 10, 90, 80);  // This line is drawn
} else {
  line(20, 80, 90, 10);  // This line is not drawn
}
```

Description Reserved word representing the logical value "true". Only variables of type **boolean** may be assigned the value **true**.

Related false
boolean

Name

try

Examples

```
BufferedReader reader;
String line;

void setup() {
  // Open the file from the createWriter() example
  reader = createReader("positions.txt");
}

void draw() {
```

```
  try {
    line = reader.readLine();
  } catch (IOException e) {
    e.printStackTrace();
    line = null;
  }
  if (line == null) {
    // Stop reading because of an error or file is empty
    noLoop();
  } else {
    String[] pieces = split(line, TAB);
    int x = int(pieces[0]);
    int y = int(pieces[1]);
    point(x, y);
  }
}
```

Description

The **try** keyword is used with **catch** to handle exceptions. Sun's Java documentation defines an exception as "an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions." This could be, for example, an error while a file is read.

Syntax

```
try {
  tryStatements
} catch (exception) {
  catchStatements
}
```

Parameters

tryStatements if this code throws an exception, then the code in "catch" is run
exception the Java exception that was thrown
catchStatements code that handles the exception

Name

void

Examples

```
void setup() {  // setup() does not return a value
  size(200, 200);
}

void draw() {  // draw() does not return a value
  line(10, 100, 190, 100);
  drawCircle();
}

void drawCircle() {  // This function also does not return a
value
  ellipse(30, 30, 50, 50);
}
```

Description

Keyword used indicate that a function returns no value. Each function must either return a value of a specific datatype or use the keyword **void** to specify it returns nothing.

Syntax

```
void function {
  statements
}
```

Parameters

function any function that is being defined or implemented
statements any valid statements

Name

cursor()

Examples

```
// Move the mouse left and right across the image
// to see the cursor change from a cross to a hand

void draw()
{
  if (mouseX < 50) {
    cursor(CROSS);
  } else {
    cursor(HAND);
  }
}
```

Description Sets the cursor to a predefined symbol or an image, or makes it visible if already hidden. If you are trying to set an image as the cursor, the recommended size is 16x16 or 32x32 pixels. It is not possible to load an image as the cursor if you are exporting your program for the Web, and not all MODES work with all browsers. The values for parameters **x** and **y** must be less than the dimensions of the image.

Setting or hiding the cursor does not generally work with "Present" mode (when running full-screen).

Syntax

```
cursor(kind)
cursor(img)
cursor(img, x, y)
cursor()
```

Parameters

kind int: either ARROW, CROSS, HAND, MOVE, TEXT, or WAIT
img PImage: any variable of type PImage
x int: the horizontal active spot of the cursor
y int: the vertical active spot of the cursor

Returns void

Related noCursor()

Name

displayHeight

Examples

```
size(displayWidth, displayHeight);
line(0, 0, width, height);
```

Description System variable that stores the height of the entire screen display. This is used to run a full-screen program on any display size.

Name

displayWidth

Examples

```
size(displayWidth, displayHeight);
line(0, 0, width, height);
```

Description System variable that stores the width of the entire screen display. This is used to run a full-screen program on any display size.

Name

focused

Examples

```
if (focused) {  // or "if (focused == true)"
  ellipse(25, 25, 50, 50);
```

```
} else {
  line(0, 0, 100, 100);
  line(100, 0, 0, 100);
}
```

Description Confirms if a Processing program is "focused," meaning that it is active and will accept mouse or keyboard input. This variable is "true" if it is focused and "false" if not.

Name

frameCount

Examples

```
void setup() {
  frameRate(30);
}

void draw() {
  line(0, 0, width, height);
  println(frameCount);
}
```

Description The system variable **frameCount** contains the number of frames that have been displayed since the program started. Inside **setup()** the value is 0, after the first iteration of draw it is 1, etc.

Related frameRate

Name

frameRate()

Examples

```
void setup() {
  frameRate(4);
}
int pos = 0;
void draw() {
  background(204);
  pos++;
  line(pos, 20, pos, 80);
  if (pos > width) {
    pos = 0;
  }
}
```

Description Specifies the number of frames to be displayed every second. For example, the function call **frameRate(30)** will attempt to refresh 30 times a second. If the processor is not fast enough to maintain the specified rate, the frame rate will not be achieved. Setting the frame rate within **setup()** is recommended. The default rate is 60 frames per second.

Syntax `frameRate(fps)`

Parameters **fps** float: number of desired frames per second

Returns void

Related setup()
draw()
loop()
noLoop()
redraw()

Name

frameRate

Examples

```
void setup() {
  frameRate(30);
}

void draw() {
  line(0, 0, width, height);
  println(frameRate);
}
```

Description The system variable **frameRate** contains the approximate frame rate of a running sketch. The initial value is 10 fps and is updated with each frame. The value is averaged over several frames, and so will only be accurate after the draw function has run 5-10 times.

Related frameRate

Name

height

Examples

```
noStroke();
background(0);
rect(40, 0, 20, height);
rect(60, 0, 20, height/2);
```

Description System variable that stores the height of the display window. This value is set by the second parameter of the **size()** function. For example, the function call **size(320, 240)** sets the **height** variable to the value 240. The value of **height** defaults to 100 if **size()** is not used in a program.

Name

noCursor()

Examples

```
// Press the mouse to hide the cursor
void draw()
{
  if (mousePressed == true) {
    noCursor();
  } else {
    cursor(HAND);
  }
}
```

Description Hides the cursor from view. Will not work when running the program in a web browser or in full screen (Present) mode.

Syntax `noCursor()`

Returns void

Related cursor()

Name

size()

```
size(200, 100);
background(153);
line(0, 0, width, height);
```

```
void setup() {
  size(320, 240);
}

void draw() {
  background(153);
  line(0, 0, width, height);
}
```

Examples

```
size(150, 200, P3D);  // Specify P3D renderer
background(153);

// With P3D, we can use z (depth) values...
line(0, 0, 0, width, height, -100);
line(width, 0, 0, width, height, -100);
line(0, height, 0, width, height, -100);

//...and 3D-specific functions, like box()
translate(width/2, height/2);
rotateX(PI/6);
rotateY(PI/6);
box(35);
```

Defines the dimension of the display window in units of pixels. The **size()** function must be the first line of code, or the first code inside **setup()**. Any code that appears before the **size()** command may run more than once, which can lead to confusing results.

The system variables **width** and **height** are set by the parameters passed to this function. If **size()** is not used, the window will be given a default size of 100x100 pixels.

The **size()** function can only be used once inside a sketch, and it cannot be used for resizing.

Description Do not use variables as the parameters to **size()** function, because it will cause problems when exporting your sketch. When variables are used, the dimensions of your sketch cannot be determined during export. Instead, employ numeric values in the **size()** statement, and then use the built-in **width** and **height** variables inside your program when the dimensions of the display window are needed.

The maximum width and height is limited by your operating system, and is usually the width and height of your actual screen. On some machines it may simply be the number of pixels on your current screen, meaning that a screen of 800x600 could support **size(1600, 300)**, since that is the same number of pixels. This varies widely, so you'll have to try different rendering modes and sizes until you get what you're looking for. If you need something larger, use

createGraphics to create a non-visible drawing surface.

The **renderer** parameter selects which rendering engine to use. For example, if you will be drawing 3D shapes, use **P3D**. In addition to the default renderer, other renderers are:

P2D (Processing 2D): A renderer that supports two-dimensional drawing.

P3D (Processing 3D): 3D graphics renderer that makes use of OpenGL-compatible graphics hardware.

PDF: The PDF renderer draws 2D graphics directly to an Acrobat PDF file. This produces excellent results when you need vector shapes for high-resolution output or printing. You must first use Import Library → PDF to make use of the library. More information can be found in the PDF library reference.

Syntax

```
size(w, h)
size(w, h, renderer)
```

Parameters

w int: width of the display window in units of pixels
h int: height of the display window in units of pixels
renderer String: Either P2D, P3D, or PDF

Returns void

Name

width

Examples

```
noStroke();
background(0);
rect(0, 40, width, 20);
rect(0, 60, width/2, 20);
```

Description System variable that stores the width of the display window. This value is set by the first parameter of the **size()** function. For example, the function call **size(320, 240)** sets the **width** variable to the value 320. The value of **width** defaults to 100 if **size()** is not used in a program.

Name

boolean

Examples

```
boolean a = false;
if (!a) {
  rect(30, 20, 50, 50);
}
a = true;
if (a) {
  line(20, 10, 90, 80);
  line(20, 80, 90, 10);
}
```

Description Datatype for the Boolean values **true** and **false**. It is common to use **boolean** values with control statements to determine the flow of a program. The first time

a variable is written, it must be declared with a statement expressing its datatype.

Syntax

```
boolean var
boolean var = booleanvalue
```

Parameters

var variable name referencing the value
booleanvalue true or false

Related

true
false

Name

byte

Examples

```
// Declare variable 'a' of type byte
byte a;

// Assign 23 to 'a'
a = 23;

// Declare variable 'b' and assign it the value -128
byte b = -128;

// Declare variable 'c' and assign it the sum of 'a' and 'b'.
// By default, when two bytes are added, they are converted
// to an integer. To keep the answer as a byte, cast them
// to a byte with the byte() conversion function
byte c = byte(a + b);
```

Description

Datatype for bytes, 8 bits of information storing numerical values from 127 to -128. Bytes are a convenient datatype for sending information to and from the serial port and for representing letters in a simpler format than the **char** datatype. The first time a variable is written, it must be declared with a statement expressing its datatype. Subsequent uses of this variable must not reference the datatype because Processing will think the variable is being declared again.

Syntax

```
byte var
byte var = value
```

Parameters

var variable name referencing the value
value a number between 127 to -128

Related

int
float
boolean

Name

char

Examples

```
char m;      // Declare variable 'm' of type char
m = 'A';     // Assign 'm' the value "A"
int n = '&'; // Declare variable 'n' and assign it the value "&"
```

Description

Datatype for characters, typographic symbols such as A, d, and $. A **char** stores letters and symbols in the Unicode format, a coding system developed to support a variety of world languages. Each **char** is two bytes (16 bits) in length and is distinguished by surrounding it with single quotes. Character escapes may also stored as a **char**. For example, the representation for the "delete" key is 127. The first time a variable is written, it must be declared with a statement expressing its datatype. Subsequent uses of this variable must not reference the datatype

because Processing will think the variable is being declared again.

Syntax

```
char var
char var = value
```

Parameters

var variable name referencing the value
value any character

Related String

Name

color

Examples

```
color c1 = color(204, 153, 0);
color c2 = #FFCC00;
noStroke();
fill(c1);
rect(0, 0, 25, 100);
fill(c2);
rect(25, 0, 25, 100);
color c3 = get(10, 50);
fill(c3);
rect(50, 0, 50, 100);
```

Description

Datatype for storing color values. Colors may be assigned with **get()** and **color()** or they may be specified directly using hexadecimal notation such as **#FFCC00** or **0xFFFFCCOO**.

Using **print()** or **println()** on a color will produce strange results (usually negative numbers) because of the way colors are stored in memory. A better technique is to use the **hex()** function to format the color data, or use the **red()**, **green()**, and **blue()** functions to get individual values and print those. The **hue()**, **saturation()**, and **brightness()** functions work in a similar fashion. To extract red, green, and blue values more quickly (for instance when analyzing an image or a frame of video), use bit shifting.

Values can also be created using web color notation. For example, "color c = #006699".

Web color notation only works for opaque colors. To define a color with an alpha value, you can either use the **color()** function, or use hexadecimal notation. For hex notation, prefix the values with "0x", for instance "color c = 0xCC006699". In that example, CC (the hex value of 204) is the alpha value, and the remainder is identical to a web color. Note the alpha value is first in the hexadecimal notation (but last when used with the **color()** function, or functions like **fill()** and **stroke()**).

From a technical standpoint, colors are 32 bits of information ordered as AAAAAAAARRRRRRRRGGGGGGGGBBBBBBBB where the A's contain the alpha value, the R's are the red value, G's are green, and B's are blue. Each component is 8 bits (a number between 0 and 255). These values can be manipulated with bit shifting.

Name

double

Examples

```
double a;             // Declare variable 'a' of type float
a = 1.5387;           // Assign 'a' the value 1.5387
double b = -2.984;    // Declare variable 'b' and assign it the
value -2.984
double c = a + b;     // Declare variable 'c' and assign it the
sum of 'a' and 'b'
float f = (float)c;   // Converts the value of 'c' from a double
to a float
```

Description

Datatype for floating-point numbers larger than those that can be stored in a **float**. A **float** is a 32-bit values that can be as large as 3.40282347E+38 and as low as -3.40282347E+38. A **double** can be used similarly to a float, but can have a greater magnitude because it's a 64-bit number. Processing functions don't use this datatype, so while they work in the language, you'll usually have to convert to a **float** using the **(float)** syntax before passing into a function.

Syntax

```
double var
double var = value
```

Parameters

var variable name referencing the float
valueany floating-point value

Related float

Name

float

```
float a;            // Declare variable 'a' of type float
a = 1.5387;         // Assign 'a' the value 1.5387
float b = -2.984;   // Declare variable 'b' and assign it the
value -2.984
float c = a + b;    // Declare variable 'c' and assign it the sum
of 'a' and 'b'
```

Examples

```
float f = 0;
for (int i = 0 ; i < 100000; i++) {
  f = f + 0.0001;  // Bad idea! See below.
}

for (int i = 0; i < 100000; i++) {
  // The variable 'f' will work better here, less affected by
rounding
  float f = i / 1000.0;  // Count by thousandths
}
```

Description

Data type for floating-point numbers, a number that has a decimal point. Floats are not precise, so avoid adding small values (such as 0.0001) may not always increment because of rounding error. If you want to increment a value in small intervals, use an int, and divide by a float value before using it. (See above example.)

Floating-point numbers can be as large as 3.40282347E+38 and as low as -3.40282347E+38. They are stored as 32 bits (4 bytes) of information. The float data type is inherited from Java, and you can read more about the technical details here or here.

Processing supports the 'double' datatype from Java as well, however it is not documented because none of the Processing functions use double values, which are overkill for nearly all work created in Processing and use more memory. We do not plan to support doubles, as it would require increasing the number of API functions significantly.

Syntax

```
float var
float var = value
```

Parameters

var variable name referencing the float
value any floating-point value

Related int

Name

int

Examples

```
int a;          // Declare variable 'a' of type int
a = 23;         // Assign 'a' the value 23
int b = -256;   // Declare variable 'b' and assign it the value
-256
int c = a + b;  // Declare variable 'c' and assign it the sum of
'a' and 'b'
```

Description

Datatype for integers, numbers without a decimal point. Integers can be as large as 2,147,483,647 and as low as -2,147,483,648. They are stored as 32 bits of information. The first time a variable is written, it must be declared with a statement expressing its datatype. Subsequent uses of this variable must not reference the datatype because Processing will think the variable is being declared again.

Syntax

```
int var
int var = value
```

Parameters

var variable name referencing the value
value any integer value

Related float

Name

long

Examples

```
long a;          // Declare variable 'a' of type long
a = 23;          // Assign 'a' the value 23
long b = -256;   // Declare variable 'b' and assign it the value
-256
long c = a + b;  // Declare variable 'c' and assign it the sum of
'a' and 'b'
int i = (int)c;  // Converts the value of 'c' from a long to an
int
```

Description

Datatype for large integers. While integers can be as large as 2,147,483,647 and as low as -2,147,483,648 (stored as 32 bits), a **long** integer has a minimum value of -9,223,372,036,854,775,808 and a maximum value of 9,223,372,036,854,775,807 (stored as 64 bits). Use this datatype when you need a number to have a greater magnitude than can be stored within an **int**. Processing functions don't use this datatype, so while they work in the language, you'll usually have to convert to a **int** using the **(int)** syntax before passing into a function.

Syntax

```
long var
long var = value
```

Parameters **var** variable name referencing the value
value any integer value

Related int

Name

Array

Examples

```
int[] numbers = new int[3];
numbers[0] = 90;  // Assign value to first element in the array
numbers[1] = 150; // Assign value to second element in the array
numbers[2] = 30;  // Assign value to third element in the array
int a = numbers[0] + numbers[1]; // Sets variable 'a' to 240
int b = numbers[1] + numbers[2]; // Sets variable 'b' to 180
```

```
int[] numbers = { 90, 150, 30 };  // Alternate syntax
int a = numbers[0] + numbers[1];  // Sets variable 'a' to 240
int b = numbers[1] + numbers[2];  // Sets variable 'b' to 180
```

```
int degrees = 360;
float[] cos_vals = new float[degrees];
// Use a for() loop to quickly iterate
// through all values in an array.
for (int i=0; i < degrees; i++) {
  cos_vals[i] = cos(TWO_PI/degrees * i);
}
```

Description

An array is a list of data. It is possible to have an array of any type of data. Each piece of data in an array is identified by an index number representing its position in the array. The first element in the array is **[0]**, the second element is **[1]**, and so on. Arrays are similar to objects, so they must be created with the keyword **new**.

Each array has a variable **length**, which is an integer value for the total number of elements in the array. Note that since index numbering begins at zero (not 1), the last value in an array with a **length** of 5 should be referenced as **array[4]** (that is, the **length** minus 1), not **array[5]**, which would trigger an error.

Another common source of confusion is the difference between using **length** to get the size of an array and **length()** to get the size of a String. Notice the presence of parentheses when working with Strings. (**array.length** is a variable, while **String.length()** is a method specific to the String class.)

Syntax

```
datatype[] var
var[element] = value
var.length
```

Parameters

datatype any primitive or compound datatype, including user-defined classes
var any valid variable name
element int: must not exceed the length of the array minus 1
value data to assign to the array element; must be the same datatype as the array

---

Name

ArrayList

Examples

```
// This is a code fragment that shows how to use an ArrayList.
// It won't compile because it's missing the Ball class.
```

```
// Declaring the ArrayList, note the use of the syntax "" to
indicate
// our intention to fill this ArrayList with Ball objects
ArrayList<Ball> balls;

void setup() {
  size(200, 200);
  balls = new ArrayList<Ball>();  // Create an empty ArrayList
  balls.add(new Ball(width/2, 0, 48));  // Start by adding one
element
}

void draw() {
  background(255);

  // With an array, we say balls.length. With an ArrayList,
  // we say balls.size(). The length of an ArrayList is dynamic.
  // Notice how we are looping through the ArrayList backwards.
  // This is because we are deleting elements from the list.
  for (int i = balls.size()-1; i >= 0; i--) {
    Ball ball = balls.get(i);
    ball.move();
    ball.display();
    if (ball.finished()) {
      // Items can be deleted with remove().
      balls.remove(i);
    }
  }
}

void mousePressed() {
  // A new ball object is added to the ArrayList, by default to
the end.
  balls.add(new Ball(mouseX, mouseY, ballWidth));
}
```

Description

An **ArrayList** stores a variable number of objects. This is similar to making an array of objects, but with an **ArrayList**, items can be easily added and removed from the ArrayList and it is resized dynamically. This can be very convenient, but it's slower than making an array of objects when using many elements. Note that for resizable lists of integers, floats, and Strings, you can use the Processing classes IntList, FloatList, and StringList.

An ArrayList is a resizable-array implementation of the Java List interface. It has many methods used to control and search its contents. For example, the length of the **ArrayList** is returned by its **size()** method, which is an integer value for the total number of elements in the list. An element is added to an **ArrayList** with the **add()** method and is deleted with the **remove()** method. The **get()** method returns the element at the specified position in the list. (See the above example for context.)

For a list of the numerous **ArrayList** features, please read the Java reference description.

Constructor

```
ArrayList<Type>()
ArrayList<Type>(initialCapacity)
```

Parameters

Type	Class Name: the data type for the objects to be placed in the ArrayList.
initialCapacity	int: defines the initial capacity of the list; it's empty by default

Related

IntList

FloatList
StringList

Name

FloatDict

Examples

```
FloatDict inventory;

void setup() {
  size(200, 200);
  inventory = new FloatDict();
  inventory.set("coffee",108.6);
  inventory.set("flour",5.8);
  inventory.set("tea",8.2);
  println(inventory);
  noLoop();
  fill(0);
  textAlign(CENTER);
}

void draw() {
  float weight = inventory.get("coffee");
  text(weight, width/2, height/2);
}
```

Description A simple class to use a String as a lookup for an float value. String "keys" are associated with floating-point values.

Methods

size()	Returns the number of key/value pairs
clear()	Remove all entries
keys()	Return the internal array being used to store the keys
keyArray()	Return a copy of the internal keys array
values()	Return the internal array being used to store the values
valueArray()	Create a new array and copy each of the values into it
get()	Return a value for the specified key
set()	Create a new key/value pair or change the value of one
hasKey()	Check if a key is a part of the data structure
add()	Add to a value
sub()	Subtract from a value
mult()	Multiply a value
div()	Divide a value
remove()	Remove a key/value pair
sortKeys()	Sort the keys alphabetically
sortKeysReverse()	Sort the keys alphabetially in reverse
sortValues()	Sort by values in ascending order
sortValuesReverse()	Sort by values in descending order

Constructor `FloatDict()`

Related
IntDict
StringDict

Name

FloatList

Examples

```
FloatList inventory;

void setup() {
```

```
  size(200, 200);
  inventory = new FloatList();
  inventory.append(108.6);
  inventory.append(5.8);
  inventory.append(8.2);
  println(inventory);
  noLoop();
  fill(0);
  textAlign(CENTER);
}

void draw() {
  float nums = inventory.get(2);
  text(nums, width/2, height/2);
}
```

Description

Helper class for a list of floats. Lists are designed to have some of the features of ArrayLists, but to maintain the simplicity and efficiency of working with arrays.

Functions like sort() and shuffle() always act on the list itself. To get a sorted copy, use list.copy().sort().

Methods

size()	Get the length of the list
clear()	Remove all entries from the list
get()	Get an entry at a particular index
set()	Set the entry at a particular index
remove()	Remove an element from the specified index
append()	Add a new entry to the list
hasValue()	Check if a number is a part of the list
add()	Add to a value
sub()	Subtract from a value
mult()	Multiply a value
div()	Divide a value
min()	Return the smallest value
max()	Return the largest value
sort()	Sorts an array, lowest to highest
sortReverse()	Reverse sort, orders values from highest to lowest
reverse()	Reverse sort, orders values by first digit
shuffle()	Randomize the order of the list elements
array()	Create a new array with a copy of all the values

Constructor `FloatList()`

Related IntList
StringList

Name

HashMap

Examples

```
import java.util.Map;

// Note the HashMap's "key" is a String and "value" is an Integer
HashMap<String,Integer> hm = new HashMap<String,Integer>();

// Putting key-value pairs in the HashMap
hm.put("Ava", 1);
hm.put("Cait", 35);
hm.put("Casey", 36);
```

```
// Using an enhanced loop to interate over each entry
for (Map.Entry me : hm.entrySet()) {
  print(me.getKey() + " is ");
  println(me.getValue());
}

// We can also access values by their key
int val = hm.get("Casey");
println("Casey is " + val);
```

Description

A **HashMap** stores a collection of objects, each referenced by a key. This is similar to an **Array**, only instead of accessing elements with a numeric index, a **String** is used. (If you are familiar with associative arrays from other languages, this is the same idea.) The above example covers basic use, but there's a more extensive example included with the Processing examples. In addition, for simple pairings of Strings and integers, Strings and floats, or Strings and Strings, you can now use the simpler IntDict, FloatDict, and StringDict classes.

For a list of the numerous **HashMap** features, please read the Java reference description.

Constructor

```
HashMap<Key, Value>()
HashMap<Key, Value>(initialCapacity)
HashMap<Key, Value>(initialCapacity, loadFactor)
HashMap<Key, Value>(m)
```

Parameters

Key	Class Name: the data type for the HashMap's keys
Value	Class Name: the data type for the HashMap's values
initialCapacity	int: defines the initial capacity of the map; the default is 16
loadFactor	float: the load factor for the map; the default is 0.75
m	Map: gives the new HashMap the same mappings as this Map

Related

IntDict
FloatDict
StringDict

Name

IntDict

Examples

```
IntDict inventory;

void setup() {
  size(200, 200);
  inventory = new IntDict();
  inventory.set("cd",84);
  inventory.set("tapes",15);
  inventory.set("records",102);
  println(inventory);
  noLoop();
  fill(0);
  textAlign(CENTER);
}

void draw() {
  int numRecords = inventory.get("records");
  text(numRecords, width/2, height/2);
}
```

Description

A simple class to use a String as a lookup for an int value. String "keys" are associated with integer values.

	size()	Returns the number of key/value pairs
	clear()	Remove all entries
	keys()	Return the internal array being used to store the keys
	keyArray()	Return a copy of the internal keys array
	values()	Return the internal array being used to store the keys
	valueArray()	Create a new array and copy each of the values into it
	get()	Return a value for the specified key
	set()	Create a new key/value pair or change the value of one
	hasKey()	Check if a key is a part of the data structure
Methods	increment()	Increase the value of a specific key value by 1
	add()	Add to a value
	sub()	Subtract from a value
	mult()	Multiply a value
	div()	Divide a value
	remove()	Remove a key/value pair
	sortKeys()	Sort the keys alphabetically
	sortKeysReverse()	Sort the keys alphabetially in reverse
	sortValues()	Sort by values in ascending order
	sortValuesReverse()	Sort by values in descending order

Constructor `IntDict()`

Related FloatDict
StringDict

Name

IntList

```
IntList inventory;

void setup() {
  size(200, 200);
  inventory = new IntList();
  inventory.append(84);
  inventory.append(15);
  inventory.append(102);
  println(inventory);
  noLoop();
  fill(0);
  textAlign(CENTER);
}

void draw() {
  int nums = inventory.get(2);
  text(nums, width/2, height/2);
}
```

Examples

Description

Helper class for a list of ints. Lists are designed to have some of the features of ArrayLists, but to maintain the simplicity and efficiency of working with arrays.

Functions like sort() and shuffle() always act on the list itself. To get a sorted copy, use list.copy().sort().

	size()	Get the length of the list
	clear()	Remove all entries from the list
Methods	get()	Get an entry at a particular index
	set()	Set the entry at a particular index

remove()	Remove an element from the specified index
append()	Add a new entry to the list
hasValue()	Check if a number is a part of the list
increment()	Add one to a value
add()	Add to a value
sub()	Subtract from a value
mult()	Multiply a value
div()	Divide a value
min()	Return the smallest value
max()	Return the largest value
sort()	Sorts the array, lowest to highest
sortReverse()	Reverse sort, orders values from highest to lowest
reverse()	Reverse sort, orders values by first digit
shuffle()	Randomize the order of the list elements
array()	Create a new array with a copy of all the values

Constructor `IntList()`

Related FloatList
StringList

Name

JSONArray

Examples

```
String[] species = { "Capra hircus", "Panthera pardus", "Equus
zebra" };
String[] names = { "Goat", "Leopard", "Zebra" };

JSONArray values;

void setup() {

  values = new JSONArray();

  for (int i = 0; i < species.length; i++) {

    JSONObject animal = new JSONObject();

    animal.setInt("id", i);
    animal.setString("species", species[i]);
    animal.setString("name", names[i]);

    values.setJSONObject(i, animal);
  }

  saveJSONArray(values, "data/new.json");
}

// Sketch saves the following to a file called "new.json":
// [
//   {
//     "id": 0,
//     "species": "Capra hircus",
//     "name": "Goat"
//   },
//   {
//     "id": 1,
//     "species": "Panthera pardus",
//     "name": "Leopard"
```

```
//   },
//   {
//     "id": 2,
//     "species": "Equus zebra",
//     "name": "Zebra"
//   }
// ]
```

Description A **JSONArray** stores an array of JSON objects. **JSONArray**s can be generated from scratch, dynamically, or using data from an existing file. JSON can also be output and saved to disk, as in the example above.

Methods

getString()	Gets the String value associated with an index
getInt()	Gets the int value associated with an index
getFloat()	Gets the float value associated with an index
getBoolean()	Gets the boolean value associated with an index
getJSONArray()	Gets the JSONArray associated with an index value
getJSONObject()	Gets the JSONObject associated with an index value
getStringArray()	Gets the entire array as an array of Strings
getIntArray()	Gets the entire array as array of ints
append()	Appends a value, increasing the array's length by one
setString()	Put a String value in the JSONArray
setInt()	Put an int value in the JSONArray
setFloat()	Put a float value in the JSONArray
setBoolean()	Put a boolean value in the JSONArray
setJSONArray()	Sets the JSONArray value associated with an index value
setJSONObject()	Sets the JSONObject value associated with an index value
size()	Gets the number of elements in the JSONArray
remove()	Removes an element

Constructor `JSONArray()`

Related
JSONObject
loadJSONObject()
loadJSONArray()
saveJSONObject()
saveJSONArray()

Name

JSONObject

Examples

```
JSONObject json;

void setup() {

  json = new JSONObject();

  json.setInt("id", 0);
  json.setString("species", "Panthera leo");
  json.setString("name", "Lion");

  saveJSONObject(json, "data/new.json");
}

// Sketch saves the following to a file called "new.json":
// {
//   "id": 0,
//   "species": "Panthera leo",
//   "name": "Lion"
```

```
// }
```

Description

A **JSONObject** stores JSON data with multiple name/value pairs. Values can be numeric, Strings, booleans, other **JSONObject**s or **JSONArray**s, or null. **JSONObject** and **JSONArray** objects are quite similar and share most of the same methods; the primary difference is that the latter stores an array of JSON objects, while the former represents a single JSON object.

JSON can be generated from scratch, dynamically, or using data from an existing file. JSON can also be output and saved to disk, as in the example above.

Methods

getString()	Gets the string value associated with a key
getInt()	Gets the int value associated with a key
getFloat()	Gets the float value associated with a key
getBoolean()	Gets the boolean value associated with a key
getJSONArray()	Gets the JSONArray value associated with a key
getJSONObject()	Gets the JSONObject value associated with a key
setString()	Put a key/String pair in the JSONObject
setInt()	Put a key/int pair in the JSONObject
setFloat()	Put a key/float pair in the JSONObject
setBoolean()	Put a key/boolean pair in the JSONObject
setJSONObject()	Sets the JSONObject value associated with a key
setJSONArray()	Sets the JSONArray value associated with a key

Related

JSONArray
loadJSONObject()
loadJSONArray()
saveJSONObject()
saveJSONArray()

Name

Object

Examples

```
// Declare and construct two objects (h1, h2) from the class
HLine
HLine h1 = new HLine(20, 2.0);
HLine h2 = new HLine(50, 2.5);

void setup()
{
  size(200, 200);
  frameRate(30);
}

void draw() {
  background(204);
  h1.update();
  h2.update();
}

class HLine {
  float ypos, speed;
  HLine (float y, float s) {
    ypos = y;
    speed = s;
  }
  void update() {
```

```
    ypos += speed;
    if (ypos > width) {
      ypos = 0;
    }
    line(0, ypos, width, ypos);
  }
}
```

Description Objects are instances of classes. A class is a grouping of related methods (functions) and fields (variables and constants).

Syntax `ClassName instanceName`

Parameters **ClassName** the class from which to create the new object
instanceName the name for the new object

Related class

Name

String

```
String str1 = "CCCP";
char data[] = {'C', 'C', 'C', 'P'};
String str2 = new String(data);
println(str1);  // Prints "CCCP" to the console
println(str2);  // Prints "CCCP" to the console
```

Examples

```
// Comparing String objects, see reference below.
String p = "potato";
if (p == "potato") {
  println("p == potato, yep.");  // This will not print
}
// The correct way to compare two Strings
if (p.equals("potato")) {
  println("Yes, the contents of p and potato are the same.");
}
```

```
// Use a backslash to include quotes in a String
String quoted = "This one has \"quotes\"";
println(quoted);  // This one has "quotes"
```

Description A string is a sequence of characters. The class **String** includes methods for examining individual characters, comparing strings, searching strings, extracting parts of strings, and for converting an entire string uppercase and lowercase. Strings are always defined inside double quotes ("Abc"), and characters are always defined inside single quotes ('A').

To compare the contents of two Strings, use the **equals()** method, as in "if (a.equals(b))", instead of "if (a == b)". A String is an Object, so comparing them with the == operator only compares whether both Strings are stored in the same memory location. Using the **equals()** method will ensure that the actual contents are compared. (The troubleshooting reference has a longer explanation.)

Because a String is defined between quotation marks, to include such marks within the String itself you must use the \ (backslash) character. (See the third example above.) This is known as an *escape sequence*. Other escape sequences include \t for the tab character and \n for new line. Because backslash is the escape character, to include a single backslash within a String, you must use two consecutive backslashes, as in: \\

There are more string methods than those linked from this page. Additional documentation is located online in the official Java documentation..

Methods

charAt()	Returns the character at the specified index
equals()	Compares a string to a specified object
indexOf()	Returns the index value of the first occurance of a character within the input string
length()	Returns the number of characters in the input string
substring()	Returns a new string that is part of the input string
toLowerCase()	Converts all the characters to lower case
toUpperCase()	Converts all the characters to upper case

Constructor

```
String(data)
String(data, offset, length)
```

Parameters

data	byte[] or char[]: either an array of bytes to be decoded into characters, or an array of characters to be combined into a string
offset	int: index of the first character
length	int: number of characters

Related

char
text()

Name

StringDict

Examples

```
StringDict inventory;

void setup() {
  size(200, 200);
  inventory = new StringDict();
  inventory.set("coffee","black");
  inventory.set("flour","white");
  inventory.set("tea","green");
  println(inventory);
  noLoop();
  fill(0);
  textAlign(CENTER);
}

void draw() {
  String s = inventory.get("tea");
  text(s, width/2, height/2);
}
```

Description

A simple class to use a String as a lookup for an String value. String "keys" are associated with String values.

Methods

size()	Returns the number of key/value pairs
clear()	Remove all entries
keys()	Return the internal array being used to store the keys
keyArray()	Return a copy of the internal keys array
values()	Return the internal array being used to store the values
valueArray()	Create a new array and copy each of the values into it
get()	Return a value for the specified key
set()	Create a new key/value pair or change the value of one
hasKey()	Check if a key is a part of the data structure
remove()	Remove a key/value pair
sortKeys()	Sort the keys alphabetically
sortKeysReverse()	Sort the keys alphabetially in reverse

	sortValues()	Sort by values in ascending order
	sortValuesReverse()	Sort by values in descending order
Constructor	`StringDict()`	
Related	IntDict FloatDict	

Name

StringList

Examples

```
StringList inventory;

void setup() {
  size(200, 200);
  inventory = new StringList();
  inventory.append("coffee");
  inventory.append("flour");
  inventory.append("tea");
  println(inventory);
  noLoop();
  fill(0);
  textAlign(CENTER);
}

void draw() {
  String item = inventory.get(2);
  text(item, width/2, height/2);
}
```

Description

Helper class for a list of Strings. Lists are designed to have some of the features of ArrayLists, but to maintain the simplicity and efficiency of working with arrays.

Functions like sort() and shuffle() always act on the list itself. To get a sorted copy, use list.copy().sort().

Methods

size()	Get the length of the list
clear()	Remove all entries from the list
get()	Get an entry at a particular index
set()	Set an entry at a particular index
remove()	Remove an element from the specified index
append()	Add a new entry to the list
hasValue()	Check if a value is a part of the list
sort()	Sorts the array in place
sortReverse()	Reverse sort, orders values from highest to lowest
reverse()	To come...
shuffle()	Randomize the order of the list elements
lower()	Make the entire list lower case
upper()	Make the entire list upper case
array()	Create a new array with a copy of all the values

Constructor	`StringList()`
Related	IntList FloatList

Name

Table

Examples

```
Table table;

void setup() {

  table = new Table();

  table.addColumn("id");
  table.addColumn("species");
  table.addColumn("name");

  TableRow newRow = table.addRow();
  newRow.setInt("id", table.lastRowIndex());
  newRow.setString("species", "Panthera leo");
  newRow.setString("name", "Lion");

  saveTable(table, "data/new.csv");
}

// Sketch saves the following to a file called "new.csv":
// id,species,name
// 0,Panthera leo,Lion
```

Description

Table objects store data with multiple rows and columns, much like in a traditional spreadsheet. Tables can be generated from scratch, dynamically, or using data from an existing file. Tables can also be output and saved to disk, as in the example above.

Methods

addColumn()	Adds a new column to a table
removeColumn()	Removes a column from a table
getColumnCount()	Gets the number of columns in a table
getRowCount()	Gets the number of rows in a table
clearRows()	Removes all rows from a table
addRow()	Adds a row to a table
removeRow()	Removes a row from a table
getRow()	Gets a row from a table
rows()	Gets multiple rows from a table
getInt()	Get an integer value from the specified row and column
setInt()	Store an integer value in the specified row and column
getFloat()	Get a float value from the specified row and column
setFloat()	Store a float value in the specified row and column
getString()	Get an String value from the specified row and column
setString()	Store a String value in the specified row and column
getStringColumn()	Gets all values in the specified column
findRow()	Finds a row that contains the given value
findRows()	Finds multiple rows that contain the given value
matchRow()	Finds a row that matches the given expression
matchRows()	Finds multiple rows that match the given expression
removeTokens()	Removes characters from the table
trim()	Trims whitespace from values

Constructor `Table()`

Related

loadTable()
saveTable()
TableRow

Name

TableRow

Examples

```
Table table;

void setup() {

  table = new Table();

  table.addColumn("number", Table.INT);
  table.addColumn("mass", Table.FLOAT);
  table.addColumn("name", Table.STRING);

  TableRow row = table.addRow();
  row.setInt("number", 8);
  row.setFloat("mass", 15.9994);
  row.setString("name", "Oxygen");

  println(row.getInt("number"));   // Prints 8
  println(row.getFloat("mass"));   // Prints 15.9994
  println(row.getString("name"));  // Prints "Oxygen
}
```

Description A **TableRow** object represents a single row of data values, stored in columns, from a table.

Methods

getString()	Get an String value from the specified column
getInt()	Get an integer value from the specified column
getFloat()	Get a float value from the specified column
setString()	Store a String value in the specified column
setInt()	Store an integer value in the specified column
setFloat()	Store a float value in the specified column

Related

Table
addRow()
removeRow()
clearRows()
getRow()
rows()

Name

XML

Examples

```
// The following short XML file called "mammals.xml" is parsed
// in the code below. It must be in the project's "data" folder.
//
// <?xml version="1.0"?>
// <mammals>
//   <animal id="0" species="Capra hircus">Goat</animal>
//   <animal id="1" species="Panthera pardus">Leopard</animal>
//   <animal id="2" species="Equus zebra">Zebra</animal>
// </mammals>
```

```
XML xml;

void setup() {
  xml = loadXML("mammals.xml");
  XML[] children = xml.getChildren("animal");

  for (int i = 0; i < children.length; i++) {
    int id = children[i].getInt("id");
    String coloring = children[i].getString("species");
    String name = children[i].getContent();
    println(id + ", " + coloring + ", " + name);
```

```
  }
}

// Sketch prints:
// 0, Capra hircus, Goat
// 1, Panthera pardus, Leopard
// 2, Equus zebra, Zebra
```

XML is a representation of an XML object, able to parse XML code. Use **loadXML()** to load external XML files and create **XML** objects.

Description

Only files encoded as UTF-8 (or plain ASCII) are parsed properly; the encoding parameter inside XML files is ignored.

Methods

getParent()	Gets a copy of the element's parent
getName()	Gets the element's full name
setName()	Sets the element's name
hasChildren()	Checks whether or not an element has any children
listChildren()	Returns the names of all children as an array
getChildren()	Returns an array containing all child elements
getChild()	Returns the child element with the specified index value or path
addChild()	Appends a new child to the element
removeChild()	Removes the specified child
getAttributeCount()	Counts the specified element's number of attributes
listAttributes()	Returns a list of names of all attributes as an array
hasAttribute()	Checks whether or not an element has the specified attribute
getString()	Gets the content of an attribute as a String
setString()	Sets the content of an attribute as a String
getInt()	Gets the content of an attribute as an int
setInt()	Sets the content of an attribute as an int
getFloat()	Gets the content of an attribute as a float
setFloat()	Sets the content of an attribute as a float
getContent()	Gets the content of an element
getIntContent()	Gets the content of an element as an int
getFloatContent()	Gets the content of an element as a float
setContent()	Sets the content of an element
format()	Formats XML data as a String
toString()	Gets XML data as a String using default formatting

Constructor `XML(name)`

Parameters **name**String: creates a node with this name

Related

loadXML()
parseXML()
saveXML()

Name

binary()

Examples

```
color c = color(255, 204, 0);
println(c);              // Prints "-13312"
println(binary(c));      // Prints
"11111111111111111100110000000000"
println(binary(c, 16)); // Prints "1100110000000000"
```

Description Converts a byte, char, int, or color to a String containing the equivalent binary notation. For example, the color value produced by **color(0, 102, 153, 255)** will convert to the String value "11111111000000000110011010011001". This function can help make your geeky debugging sessions much happier.

Note that the maximum number of digits is 32, because an **int** value can only represent up to 32 bits. Specifying more than 32 digits will have no effect.

Syntax
```
binary(value)
binary(value, digits)
```

Parameters **value** char, byte, or int: value to convert
digits int: number of digits to return

Returns String

Related unbinary()
hex()
unhex()

Name

boolean()

Examples
```
String s = "true";
boolean b = boolean(s);
if (b) {
  println("The boolean is true");
} else {
  println("The boolean is false");
}
```

Description Converts an int, string, or array to its boolean representation. For int calues, the number 0 evaluates to false and all other numbers evaluate to true.

Name

byte()

Examples
```
char c = 'E';
byte b = byte(c);
println(c + " : " + b);  // Prints "E : 69"

int i = 130;
b = byte(i);
println(i + " : " + b);  // Prints "130 : -126"
```

Description Converts a primitive datatype or array to its byte representation. A byte can only be a whole number between -128 and 127, therefore when a number outside this range is converted, its value wraps to the corresponding byte representation.

Name

char()

Examples
```
int i = 65;
char c = char(i);
println(i + " : " + c);  // Prints "65 : A"

byte b = 65;
```

```
c = char(b);
println(b + " : " + c);  // Prints "65 : A"
```

Description Converts a primitive datatype or array to a numeric character representation.

Name

float()

Examples

```
int i = 65;
float f = float(i);
println(i + " : " + f);  // Prints "65 : 65.0"
```

Description Converts an int, string, or array to its floating point representation.

Name

hex()

Examples

```
color c = #ffcc00;
println(c);           // Prints "-13312"
println(hex(c));      // Prints "FFFFCC00"
println(hex(c, 6));   // Prints "FFCC00"
```

```
color c = color(255, 204, 0);
println(c);           // Prints "-13312"
println(hex(c));      // Prints "FFFFCC00"
println(hex(c, 6));   // Prints "FFCC00"
```

Description Converts a byte, char, int, or color to a String containing the equivalent hexadecimal notation. For example, the color value produced by **color(0, 102, 153)** will convert to the String value "FF006699". This function can help make your geeky debugging sessions much happier.

Note that the maximum number of digits is 8, because an **int** value can only represent up to 32 bits. Specifying more than 8 digits will not increase the length of the string further.

Syntax

```
hex(value)
hex(value, digits)
```

Parameters

value int, char, or byte: the value to convert
digits int: the number of digits (maximum 8)

Returns String

Related

unhex()
binary()
unbinary()

Name

int()

Examples

```
float f = 65.0;
int i = int(f);
println(f + " : " + i);  // Prints "65.0 : 65"
```

```
char c = 'E';
i = int(c);
println(c + " : " + i);  // Prints "E : 69"
```

Description Converts a primitive datatype, string, or array to its integer representation.

Name

str()

Examples

```
boolean b = false;
byte y = -28;
char c = 'R';
float f = -32.6;
int i = 1024;

String sb = str(b);
String sy = str(y);
String sc = str(c);
String sf = str(f);
String si = str(i);

sb = sb + sy + sc + sf + si;

println(sb);  // Prints 'false-28R-32.61024'
```

Description Returns the string representation of primitive datatypes and arrays. For example the integer 3 will return the string "3", the float -12.6 will return the string "-12.6", and a boolean value true will return the string "true".

Name

unbinary()

Examples

```
String s1 = "00010000";
String s2 = "00001000";
String s3 = "00000100";
println(unbinary(s1));  // Prints "16"
println(unbinary(s2));  // Prints "8"
println(unbinary(s3));  // Prints "4"
```

Description Converts a String representation of a binary number to its equivalent integer value. For example, **unbinary("00001000")** will return 8.

Syntax `unbinary(value)`

Parameters **value** String: String to convert to an integer

Returns int

Related binary()
hex()
unhex()

Name

unhex()

Examples

```
String hs = "FF006699";
int hi = unhex(hs);
fill(hi);
rect(30, 20, 55, 55);
```

Description Converts a String representation of a hexadecimal number to its equivalent integer value.

Syntax `unhex(value)`

Parameters **value** String: String to convert to an integer

Returns int

Related hex()
binary()
unbinary()

Name

join()

Examples

```
String[] animals = new String[3];
animals[0] = "cat";
animals[1] = "seal";
animals[2] = "bear";
String joinedAnimals = join(animals, " : ");
println(joinedAnimals);  // Prints "cat : seal : bear"
```

```
// Joining an array of ints requires first
// converting to an array of Strings
int[] numbers = new int[3];
numbers[0] = 8;
numbers[1] = 67;
numbers[2] = 5;
String joinedNumbers = join(nf(numbers, 0), ", ");
println(joinedNumbers);  // Prints "8, 67, 5"
```

Description Combines an array of Strings into one String, each separated by the character(s) used for the **separator** parameter. To join arrays of ints or floats, it's necessary to first convert them to Strings using **nf()** or **nfs()**.

Syntax

```
join(list, separator)
```

Parameters

list String[]: array of Strings
separator char: char or String to be placed between each item

Returns String

Related split()
trim()
nf()
nfs()

Name

match()

Examples

```
String s = "Inside a tag, you will find <tag>content</tag>.";
String[] m = match(s, "<tag>(.*?)</tag>");
println("Found '" + m[1] + "' inside the tag.");
// Prints to the console:
// "Found 'content' inside the tag."
```

```
String s1 = "Have you ever heard of a thing called fluoridation.
";
       s1 += "Fluoridation of water?";
String s2 = "Uh? Yes, I-I have heard of that, Jack, yes. Yes.";

String[] m1 = match(s1, "fluoridation");
if (m1 != null) {  // If not null, then a match was found
  // This will print to the console, since a match was found.
  println("Found a match in '" + s1 + "'");
} else {
```

```
  println("No match found in '" + s1 + "'");
}

String[] m2 = match(s2, "fluoridation");
if (m2 != null) {
  println("Found a match in '" + s2 + "'");
} else {
  // This will print to the console, since no match was found.
  println("No match found in '" + s2 + "'");
}
```

Description

This function is used to apply a regular expression to a piece of text, and return matching groups (elements found inside parentheses) as a String array. If there are no matches, a null value will be returned. If no groups are specified in the regular expression, but the sequence matches, an array of length 1 (with the matched text as the first element of the array) will be returned.

To use the function, first check to see if the result is null. If the result is null, then the sequence did not match at all. If the sequence did match, an array is returned.

If there are groups (specified by sets of parentheses) in the regular expression, then the contents of each will be returned in the array. Element [0] of a regular expression match returns the entire matching string, and the match groups start at element [1] (the first group is [1], the second [2], and so on).

The syntax can be found in the reference for Java's Pattern class. For regular expression syntax, read the Java Tutorial on the topic.

Syntax

```
match(str, regexp)
```

Parameters

str String: the String to be searched
regexp String: the regexp to be used for matching

Returns String[]

Related

matchAll()
split()
splitTokens()
join()
trim()

Name

matchAll()

Examples

```
String s = "Inside tags, you will find <tag>multiple</tag> ";
       s += "<tag>pieces</tag> of <tag>content</tag>.";

String[][] m = matchAll(s, "<tag>(.*?)</tag>");
for (int i = 0; i < m.length; i++) {
  println("Found '" + m[i][1] + "' inside a tag.");
}

// Prints to the console:
// "Found 'multiple' inside a tag."
// "Found 'pieces' inside a tag."
// "Found 'content' inside a tag."
```

Description

This function is used to apply a regular expression to a piece of text, and return a list of matching groups (elements found inside parentheses) as a two-dimensional String array. If there are no matches, a null value will be returned. If no groups

are specified in the regular expression, but the sequence matches, a two dimensional array is still returned, but the second dimension is only of length one.

To use the function, first check to see if the result is null. If the result is null, then the sequence did not match at all. If the sequence did match, a 2D array is returned.

If there are groups (specified by sets of parentheses) in the regular expression, then the contents of each will be returned in the array. Assuming a loop with counter variable i, element [i][0] of a regular expression match returns the entire matching string, and the match groups start at element [i][1] (the first group is [i][1], the second [i][2], and so on).

The syntax can be found in the reference for Java's Pattern class. For regular expression syntax, read the Java Tutorial on the topic.

Syntax `matchAll(str, regexp)`

Parameters **str** String: the String to be searched
regexp String: the regexp to be used for matching

Returns String[][]

Related match()
split()
splitTokens()
join()
trim()

Name

nf()

```
int a=200, b=40, c=90;
String sa = nf(a, 10);
println(sa);  // Prints "0000000200"
String sb = nf(b, 5);
println(sb);  // Prints "00040"
String sc = nf(c, 3);
println(sc);  // Prints "090"
```

Examples

```
float d = 200.94, e = 40.2, f = 9.012;
String sd = nf(d, 10, 4);
println(sd);  // Prints "0000000200.9400"
String se = nf(e, 5, 3);
println(se);  // Prints "00040.200"
String sf = nf(f, 3, 5);
println(sf);  // Prints "009.01200"
```

Description Utility function for formatting numbers into strings. There are two versions: one for formatting floats, and one for formatting ints. The values for the **digits**, **left**, and **right** parameters should always be positive integers.

As shown in the above example, **nf()** is used to add zeros to the left and/or right of a number. This is typically for aligning a list of numbers. To *remove* digits from a floating-point number, use the **int()**, **ceil()**, **floor()**, or **round()** functions.

Syntax
```
nf(num, digits)
nf(num, left, right)
```

Parameters **num** float[], int[], or int: the number(s) to format

digits int: number of digits to pad with zero
left int: number of digits to the left of the decimal point
right int: number of digits to the right of the decimal point

Returns String or String[]

Related nfs()
nfp()
nfc()

Name

nfc()

Examples

```
int i = 500000;
String si = nfc(i);
println(si);  // Prints "500,000"
float f = 42525.34343;
String fi = nfc(f, 2);
println(fi);  // Prints "42,525.34"
```

Description Utility function for formatting numbers into strings and placing appropriate commas to mark units of 1000. There are two versions: one for formatting ints, and one for formatting an array of ints. The value for the **right** parameter should always be a positive integer.

For a non-US locale, this will insert periods instead of commas, or whatever is apprioriate for that region.

Syntax

```
nfc(num)
nfc(num, right)
```

Parameters
num float[], or int[]: the number(s) to format
right int: number of digits to the right of the decimal point

Returns String[]

Related nf()
nfp()
nfc()

Name

nfp()

Examples

```
int a=200, b=-40, c=90;
String sa = nfp(a, 10);
println(sa);  // Prints "+0000000200"
String sb = nfp(b, 5);
println(sb);  // Prints "-00040"
String sc = nfp(c, 3);
println(sc);  // Prints "+090"

float d = -200.94, e = 40.2, f = -9.012;
String sd = nfp(d, 10, 4);
println(sd);  // Prints "-0000000200.9400"
String se = nfp(e, 5, 3);
println(se);  // Prints "+00040.200"
String sf = nfp(f, 3, 5);
println(sf);  // Prints "-009.01200"
```

Description Utility function for formatting numbers into strings. Similar to **nf()** but puts a "+"

in front of positive numbers and a "-" in front of negative numbers. There are two versions: one for formatting floats, and one for formatting ints. The values for the **digits**, **left**, and **right** parameters should always be positive integers.

Syntax

```
nfp(num, digits)
nfp(num, left, right)
```

Parameters

num float[], int[], or int: the number(s) to format
digits int: number of digits to pad with zeroes
left int: the number of digits to the left of the decimal point
right int: the number of digits to the right of the decimal point

Returns String or String[]

Related nf()
nfs()
nfc()

Name

nfs()

Examples

```
int a=200, b=-40, c=90;
String sa = nfs(a, 10);
println(sa);  // Prints " 0000000200"
String sb = nfs(b, 5);
println(sb);  // Prints "-00040"
String sc = nfs(c, 3);
println(sc);  // Prints " 090"

float d = -200.94, e = 40.2, f = -9.012;
String sd = nfs(d, 10, 4);
println(sd);  // Prints "-0000000200.9400"
String se = nfs(e, 5, 3);
println(se);  // Prints "00040.200"
String sf = nfs(f, 3, 5);
println(sf);  // Prints "-009.01200"
```

Description Utility function for formatting numbers into strings. Similar to **nf()**, but leaves a blank space in front of positive numbers so they align with negative numbers in spite of the minus symbol. There are two versions: one for formatting floats, and one for formatting ints. The values for the **digits**, **left**, and **right** parameters should always be positive integers.

Syntax

```
nfs(num, digits)
nfs(num, left, right)
```

Parameters

num float[], int[], or int: the number(s) to format
digits int: number of digits to pad with zeroes
left int: the number of digits to the left of the decimal point
right int: the number of digits to the right of the decimal point

Returns String or String[]

Related nf()
nfp()
nfc()

Name

split()

Examples

```
String men = "Chernenko,Andropov,Brezhnev";
String[] list = split(men, ',');
```

```
// list[0] is now "Chernenko", list[1] is "Andropov"...
```

```
String numbers = "8 67 5 309";
int[] nums = int(split(numbers, ' '));
// nums[0] is now 8, nums[1] is now 67...
```

```
String men = "Chernenko ] Andropov ] Brezhnev";
String[] list = split(men, " ] ");
// list[0] is now "Chernenko", list[1] is "Andropov"...
```

The **split()** function breaks a String into pieces using a character or string as the delimiter. The **delim** parameter specifies the character or characters that mark the boundaries between each piece. A String[] array is returned that contains each of the pieces.

Description If the result is a set of numbers, you can convert the String[] array to to a float[] or int[] array using the datatype conversion functions **int()** and **float()**. (See the second example above.)

The **splitTokens()** function works in a similar fashion, except that it splits using a range of characters instead of a specific character or sequence.

Syntax `split(value, delim)`

Parameters **value** String: the String to be split
delim char: the character or String used to separate the data

Returns String[]

Name

splitTokens()

```
String t = "a b";
String[] q = splitTokens(t);
println(q[0]);  // Prints "a"
println(q[1]);  // Prints "b"
```

Examples

```
// Despite the bad formatting, the data is parsed correctly.
// The ", " as delimiter means to break whenever a comma *or*
// a space is found in the String. Unlike the split() function,
// multiple adjacent delimiters are treated as a single break.
String s = "a, b c ,,d ";
String[] q = splitTokens(s, ", ");
println(q.length + " values found");  // Prints "4 values found"
println(q[0]);  // Prints "a"
println(q[1]);  // Prints "b"
println(q[2]);  // Prints "c"
println(q[3]);  // Prints "d"
```

The **splitTokens()** function splits a String at one or many character delimiters or "tokens." The **delim** parameter specifies the character or characters to be used as a boundary.

Description If no **delim** characters are specified, any whitespace character is used to split. Whitespace characters include tab (\t), line feed (\n), carriage return (\r), form feed (\f), and space.

After using this function to parse incoming data, it is common to convert the data from Strings to integers or floats by using the datatype conversion functions **int()**

and **float()**.

Syntax
```
splitTokens(value)
splitTokens(value, delim)
```

Parameters
value String: the String to be split
delim String: list of individual characters that will be used as separators

Returns String[]

Related
split()
join()
trim()

Name

trim()

Examples
```
String s1 = "    Somerville MA ";
println(s1);  // Prints "    Somerville MA "
String s2 = trim(s1);
println(s2);  // Prints "Somerville MA"

String[] a1 = { " inconsistent ", " spacing" };  // Note spaces
String[] a2 = trim(a1);
println(a2);
// Prints the following array contents to the console:
// [0] "inconsistent"
// [1] "spacing"
```

Description
Removes whitespace characters from the beginning and end of a String. In addition to standard whitespace characters such as space, carriage return, and tab, this function also removes the Unicode "nbsp" character.

Syntax
```
trim(str)
trim(array)
```

Parameters
str String: any string
array String[]: a String array

Returns String or String[]

Related
split()
join()

Name

append()

Examples
```
String[] sa1 = { "OH", "NY", "CA"};
String[] sa2 = append(sa1, "MA");
println(sa2);
// Prints updated array contents to the console:
// [0] "OH"
// [1] "NY"
// [2] "CA"
// [3] "MA"
```

Description
Expands an array by one element and adds data to the new position. The datatype of the **element** parameter must be the same as the datatype of the array.

When using an array of objects, the data returned from the function must be cast to the object array's data type. For example: *SomeClass[] items = (SomeClass[]) append(originalArray, element)*

Syntax
```
append(array, value)
```

Parameters	**array** Object, String[], float[], int[], char[], or byte[]: array to append
	value Object, String, float, int, char, or byte: new data for the array
Returns	byte[], char[], int[], float[], String[], or Object
Related	shorten() expand()

Name

arrayCopy()

```
String[] north = { "OH", "IN", "MI" };
String[] south = { "GA", "FL", "NC" };
arrayCopy(north, south);
println(south);
// Prints updated array contents to the console:
// [0] "OH"
// [1] "IN"
// [2] "MI"
```

Examples

```
String[] north = { "OH", "IN", "MI"};
String[] south = { "GA", "FL", "NC"};
arrayCopy(north, 1, south, 0, 2);
println(south);
// Prints updated array contents to the console:
// [0] "IN"
// [1] "MI"
// [2] "NC"
```

Description

Copies an array (or part of an array) to another array. The **src** array is copied to the **dst** array, beginning at the position specified by **srcPosition** and into the position specified by **dstPosition**. The number of elements to copy is determined by **length**. Note that copying values overwrites existing values in the destination array. To append values instead of overwriting them, use **concat()**.

The simplified version with only two arguments — **arrayCopy(src, dst)** — copies an entire array to another of the same size. It is equivalent to **arrayCopy(src, 0, dst, 0, src.length)**.

Using this function is far more efficient for copying array data than iterating through a **for()** loop and copying each element individually.

This function only copies references, which means that for most purposes it only copies one-dimensional arrays (a single set of brackets). If used with a two (or three or more) dimensional array, it will only copy the references at the first level, because a two dimensional array is simply an "array of arrays". This does not produce an error, however, because this is often the desired behavior.

Internally, this function calls Java's System.arraycopy() method, so most things that apply there are inherited.

Syntax

```
arrayCopy(src, srcPosition, dst, dstPosition, length)
arrayCopy(src, dst, length)
arrayCopy(src, dst)
```

Parameters	**src** Object: the source array
	srcPosition int: starting position in the source array
	dst Object: the destination array of the same data type as the source array
	dstPosition int: starting position in the destination array

length int: number of array elements to be copied

Returns void

Related concat()

Name

concat()

Examples

```
String[] sa1 = { "OH", "NY", "CA"};
String[] sa2 = { "KY", "IN", "MA"};
String[] sa3 = concat(sa1, sa2);
println(sa3);
// Prints updated array contents to the console:
// [0] "OH"
// [1] "NY"
// [2] "CA"
// [3] "KY"
// [4] "IN"
// [5] "MA"
```

Description

Concatenates two arrays. For example, concatenating the array { 1, 2, 3 } and the array { 4, 5, 6 } yields { 1, 2, 3, 4, 5, 6 }. Both parameters must be arrays of the same datatype.

When using an array of objects, the data returned from the function must be cast to the object array's data type. For example: *SomeClass[] items = (SomeClass[]) concat(array1, array2)*.

Syntax

```
concat(a, b)
```

Parameters

a Object, String[], float[], int[], char[], byte[], or boolean[]: first array to concatenate

b Object, String[], float[], int[], char[], byte[], or boolean[]: second array to concatenate

Returns boolean[], byte[], char[], int[], float[], String[], or Object

Related splice()
arrayCopy()

Name

expand()

Examples

```
int[] data = {0, 1, 3, 4};
println(data.length);  // Prints "4"
data = expand(data);
println(data.length);  // Prints "8"
data = expand(data, 512);
println(data.length);  // Prints "512"
```

```
PImage[] imgs = new PImage[32];
println(imgs.length);  // Prints "32"
imgs = (PImage[]) expand(imgs);
println(imgs.length);  // Prints "64"
```

Description

Increases the size of an array. By default, this function doubles the size of the array, but the optional **newSize** parameter provides precise control over the increase in size.

When using an array of objects, the data returned from the function must be cast to the object array's data type. For example: *SomeClass[] items = (SomeClass[]) expand(originalArray)*

Syntax
```
expand(list)
expand(list, newSize)
```

Parameters
list Object, String[], double[], float[], long[], int[], char[], byte[], or boolean[]: the array to expand
newSize int: new size for the array

Returns boolean[], byte[], char[], int[], long[], float[], double[], String[], or Object

Related shorten()

Name

reverse()

Examples
```
String sa[] = { "OH", "NY", "MA", "CA"};
sa = reverse(sa);
println(sa);
// Prints updated array contents to the console:
// [0] "CA"
// [1] "MA"
// [2] "NY"
// [3] "OH"
```

Description Reverses the order of an array.

Syntax
```
reverse(list)
```

Parameters
list Object, String[], float[], int[], char[], byte[], or boolean[]: booleans[], bytes[], chars[], ints[], floats[], or Strings[]

Returns boolean[], byte[], char[], int[], float[], String[], or Object

Related sort()

Name

shorten()

Examples
```
String[] sa1 = { "OH ", "NY ", "CA "};
String[] sa2 = shorten(sa1);
println(sa1);  // 'sa1' still contains OH, NY, CA
println(sa2);  // 'sa2' now contains OH, NY
```
Decreases an array by one element and returns the shortened array.

Description When using an array of objects, the data returned from the function must be cast to the object array's data type. For example: *SomeClass[] items = (SomeClass[]) shorten(originalArray)*

Syntax
```
shorten(list)
```

Parameters
list Object, String[], float[], int[], char[], byte[], or boolean[]: array to shorten

Returns boolean[], byte[], char[], int[], float[], String[], or Object

Related append()
expand()

Name

sort()

Examples

```
float[] a = { 3.4, 3.6, 2, 0, 7.1 };
a = sort(a);
println(a);
// Prints the contents of the sorted array:
// [0] 0.0
// [1] 2.0
// [2] 3.4
// [3] 3.6
// [4] 7.1
```

Examples

```
String[] s = { "deer", "elephant", "bear", "aardvark", "cat" };
s = sort(s);
println(s);
// Prints the contents of the sorted array:
// [0] "aardvark"
// [1] "bear"
// [2] "cat"
// [3] "deer"
// [4] "elephant"
```

```
String[] s = { "deer", "elephant", "bear", "aardvark", "cat" };
s = sort(s, 3);
println(s);
// Prints the contents of the array, with the first 3 elements
sorted:
// [0] "bear"
// [1] "deer"
// [2] "elephant"
// [3] "aardvark"
// [4] "cat"
```

Description Sorts an array of numbers from smallest to largest, or puts an array of words in alphabetical order. The original array is not modified; a re-ordered array is returned. The **count** parameter states the number of elements to sort. For example, if there are 12 elements in an array and **count** is set to 5, only the first 5 elements in the array will be sorted.

Syntax

```
sort(list)
sort(list, count)
```

Parameters

list String[], float[], int[], char[], or byte[]: array to sort

count int: number of elements to sort, starting from 0

Returns byte[], char[], int[], float[], or String[]

Related reverse()

Name

splice()

Examples

```
String[] a = { "OH", "NY", "CA" };
a = splice(a, "KY", 1);  // Splice one value into an array
println(a);
// Prints the following array contents to the console:
// [0] "OH"
// [1] "KY"
// [2] "NY"
// [3] "CA"

println();  // Prints a blank line

String[] b = { "VA", "CO", "IL" };
```

```
a = splice(a, b, 2);  // Splice one array of values into another
println(a);
// Prints the following array contents to the console:
// [0] "OH"
// [1] "KY"
// [2] "VA"
// [3] "CO"
// [4] "IL"
// [5] "NY"
// [6] "CA"
```

Description

Inserts a value or an array of values into an existing array. The first two parameters must be arrays of the same datatype. The first parameter specifies the intial array to be modified, and the second parameter defines the data to be inserted. The third parameter is an index value which specifies the array position from which to insert data. (Remember that array index numbering starts at zero, so the first position is 0, the second position is 1, and so on.)

When splicing an array of objects, the data returned from the function must be cast to the object array's data type. For example: *SomeClass[] items = (SomeClass[]) splice(array1, array2, index)*

Syntax

```
splice(list, value, index)
```

Parameters

list Object, String[], float[], int[], char[], byte[], or boolean[]: array to splice into

value Object, String[], String, float[], float, int[], int, char[], char, byte[], byte, boolean[], or boolean: value to be spliced in

index int: position in the array from which to insert data

Returns

boolean[], byte[], char[], int[], float[], String[], or Object

Related

concat()
subset()

Name

subset()

Examples

```
String[] sa1 = { "OH", "NY", "CA", "VA", "CO", "IL" };
String[] sa2 = subset(sa1, 1);
println(sa2);
// Prints the following array contents to the console:
// [0] "NY"
// [1] "CA"
// [2] "VA"
// [3] "CO"
// [4] "IL"
println();
String[] sa3 = subset(sa1, 2, 3);
println(sa3);
// Prints the following array contents to the console:
// [0] "CA"
// [1] "VA"
// [2] "CO"
```

Description

Extracts an array of elements from an existing array. The **list** parameter defines the array from which the elements will be copied, and the **start** and **count** parameters specify which elements to extract. If no **count** is given, elements will be extracted from the **start** to the end of the array. When specifying the **start**, remember that the first array element is 0. This function does not change the source array.

When using an array of objects, the data returned from the function must be cast to the object array's data type. For example: *SomeClass[] items = (SomeClass[]) subset(originalArray, 0, 4)*

Syntax

```
subset(list, start)
subset(list, start, count)
```

Parameters

list Object, String[], float[], int[], char[], byte[], or boolean[]: array to extract from
start int: position to begin
count int: number of values to extract

Returns boolean[], byte[], char[], int[], float[], String[], or Object

Related splice()

Name

!= (inequality)

Examples

```
int a = 22;
int b = 23;
if (a != b) {
  println("variable a is not equal to variable b");
}
```

Description Determines if one expression is not equivalent to another.

Syntax

```
value1 != value2
```

Parameters

value1 int, float, char, byte, boolean, String
value2 int, float, char, byte, boolean, String

Related

> (greater than)
< (less than)
>= (greater than or equal to)
<= (less than or equal to)
== (equality)

Name

< (less than)

Examples

```
int a = 22;
int b = 23;
if (a < b) {
  println("variable a is less then variable b ");
}
```

Description Tests if the value on the left is smaller than the value on the right.

Syntax

```
value1 < value2
```

Parameters

value1 int or float
value2 int or float

Related

> (greater than)
>= (greater than or equal to)
<= (less than or equal to)
== (equality)
!= (inequality)

Name

<= (less than or equal to)

Examples

```
int a = 22;
int b = 23;
if (a <= b) {
  println("variable a is less or equal to variable b ");
}
```

Description Tests if the value on the left is less than the value on the right or if the values are equivalent.

Syntax `value1 <= value2`

Parameters
value1 int or float
value2 int or float

Related
> (greater than)
< (less than)
>= (greater than or equal to)
== (equality)
!= (inequality)

Name

== (equality)

Examples

```
int a = 23;
int b = 23;
if (a == b) {
  println("variables a and b are equal");
}
```

Determines if two values are equivalent. The equality operator is different from the assignment operator.

Description Note that when comparing String objects, you must use the **equals()** method instead of == to compare their contents. See the reference for String or the troubleshooting note for more explanation.

Syntax `value1 == value2`

Parameters
value1 int, float, char, byte, boolean
value2 int, float, char, byte, boolean

Related
> (greater than)
< (less than)
>= (greater than or equal to)
<= (less than or equal to)
!= (inequality)

Name

> (greater than)

Examples

```
int a = 5;
int b = 13;
if (b > a) {
  println("variable b is larger the variable a");
}
```

Description Tests if the value on the left is larger than the value on the right.

Syntax `value1 > value2`

Parameters
value1 int or float
value2 int or float

Related
< (less than)
>= (greater than or equal to)
<= (less than or equal to)
== (equality)
!= (inequality)

Name

>= (greater than or equal to)

Examples

```
int a = 23;
int b = 23;
if (a >= b) {
  println("variable a is greater or equal to variable b ")
}
```

Description Tests if the value on the left is larger than the value on the right or if the values are equivalent.

Syntax `value1 >= value2`

Parameters
value1 int or float
value2 int or float

Related
> (greater than)
< (less than)
<= (less than or equal to)
== (equality)
!= (inequality)

Name

for

Examples

```
for (int i = 0; i < 40; i = i+1) {
  line(30, i, 80, i);
}
```

```
for (int i = 0; i < 80; i = i+5) {
  line(30, i, 80, i);
}
```

```
for (int i = 40; i < 80; i = i+5) {
  line(30, i, 80, i);
}
```

```
// Nested for() loops can be used to
// generate two-dimensional patterns
for (int i = 30; i < 80; i = i+5) {
  for (int j = 0; j < 80; j = j+5) {
    point(i, j);
  }
```

```
}
```

```
int[] nums = { 5, 4, 3, 2, 1 };

for (int i : nums) {
  println(i);
}
```

Controls a sequence of repetitions. A basic **for** structure has three parts: **init**, **test**, and **update**. Each part must be separated by a semicolon (;). The loop continues until the **test** evaluates to **false**. When a **for** structure is executed, the following sequence of events occurs:

1. The init statement is run.
2. The test is evaluated to be true or false.
3. If the test is *true*, jump to step 4. If the test is *false*, jump to step 6.
4. Run the statements within the block.
5. Run the update statement and jump to step 2.
6. Exit the loop.

Description

In the first example above, the **for** structure is executed 40 times. In the init statement, the value *i* is created and set to zero. *i* is less than 40, so the test evaluates as *true*. At the end of each loop, *i* is incremented by one. On the 41st execution, the test is evaluated as *false*, because *i* is then equal to 40, so *i* < *40* is no longer true. Thus, the loop exits.

A second type of **for** structure makes it easier to iterate over each element of an array. The last example above shows how it works. Within the parentheses, first define the datatype of the array, then define a variable name. This variable name will be assigned to each element of the array in turn as the **for** moves through the entire array. Finally, after the colon, define the array name to be used.

Syntax

```
for (init; test; update) {
  statements
}
```

```
for (datatype element : array) {
  statements
}
```

init	statement executed once when beginning loop
test	if the test evaluates to *true*, the statements execute
update	executes at the end of each iteration

Parameters

statements	collection of statements executed each time through the loop
datatype	datatype of elements in the array
element	temporary name to use for each element of the array
array	name of the array to iterate through

Related while

Name

while

Examples

```
int i = 0;
while (i < 80) {
  line(30, i, 80, i);
  i = i + 5;
}
```

Controls a sequence of repetitions. The **while** structure executes a series of statements continuously while the **expression** is **true**. The expression must be updated during the repetitions or the program will never "break out" of **while**.

Description This function can be dangerous because the code inside the **while** loop will not finish until the expression inside **while** becomes false. It will lock out all other code from running (e.g., mouse and keyboard events will not be updated). Be careful — if used incorrectly, this can lock up your code (and sometimes even the Processing environment itself).

Syntax

```
while (expression) {
  statements
}
```

Parameters
expression a valid expression
statements one or more statements

Related for

Name

?: (conditional)

Examples

```
int s = 0;
for (int i = 5; i < 100; i += 5) {
  s = (i < 50) ? 0 : 255;
  stroke(s);
  line(30, i, 80, i);
}
```

A shortcut for writing an **if** and **else** structure. The conditional operator, **?:** is sometimes called the ternary operator, an operator that takes three arguments. If the **test** evaluates to **true**, **expression1** is evaluated and returned. If the **condition** evaluates to **false**, **expression2** is evaluated and returned.

The following conditional expression:

```
result = test ? expression1 : expression2
```

Description is equivalent to this structure:

```
if (test) {

  result = expression1

} else {

  result = expression2

}
```

Syntax

```
test ? expression1 : expression2
```

Parameters
test any valid expression which evaluates to true or false
expression1 any valid expression
expression2 any valid expression

Related if

else

Name

break

Examples

```
char letter = 'B';

switch(letter) {
  case 'A':
    println("Alpha");  // Does not execute
    break;
  case 'B':
    println("Bravo");  // Prints "Bravo"
    break;
  default:
    println("Zulu");   // Does not execute
    break;
}
```

Description Ends the execution of a structure such as **switch**, **for**, or **while** and jumps to the next statement after.

Related
switch
for
while

Name

case

Examples

```
char letter = 'B';

switch(letter) {
  case 'A':
    println("Alpha");  // Does not execute
    break;
  case 'B':
    println("Bravo");  // Prints "Bravo"
    break;
  default:
    println("Zulu");   // Does not execute
    break;
}
```

Description Denotes the different labels to be evaluated with the parameter in the **switch** structure.

Syntax

```
case label: statements
```

Parameters

label byte, char, or int

statements one or more valid statements

Related
switch
default
break

Name

continue

Examples

```
for (int i = 0; i < 100; i += 10) {
```

```
  if (i == 70) {  // If 'i' is 70,
    continue;     // skip to the next iteration,
  }               // therefore not drawing the line.
  line(i, 0, i, height);
}
```

Description When run inside of a **for** or **while**, it skips the remainder of the block and starts the next iteration.

Syntax `continue`

Related for
while

Name

default

Examples

```
char letter = 'F';

switch(letter) {
  case 'A':
    println("Alpha");  // Does not execute
    break;
  case 'B':
    println("Bravo");  // Does not execute
    break;
  default:
    println("Zulu");   // Prints "Zulu"
    break;
}
```

Description Keyword for defining the default condition of a **switch**. If none of the case labels match the **switch** parameter, the statement(s) after the **default** syntax are executed. Switch structures don't require a **default**.

Syntax `default: statements`

Parameters **statements** one or more valid statements to be executed

Related switch
break
case

Name

else

Examples

```
for (int i = 5; i < 95; i += 5) {
  if (i < 35) {
    line(30, i, 80, i);
  } else {
    line(20, i, 90, i);
  }
}
```

```
for (int i = 5; i < 95; i += 5) {
  if (i < 35) {
    line(30, i, 80, i);
  } else if (i < 65) {
    line(20, i, 90, i);
```

```
  } else {
    line(0, i, 100, i);
  }
}
```

Description Extends the **if** structure allowing the program to choose between two or more block of code. It specifies a block of code to execute when the expression in **if** is **false**.

Syntax

```
if (expression) {
  statements
} else {
  statements
}
```

```
if (expression) {
  statements
} else if (expression) {
  statements
} else {
  statements
}
```

Parameters
expression any valid expression that evaluates to true or false
statements one or more statements to be executed

Related if

Name

if

Examples

```
for (int i = 5; i < height; i += 5) {
  stroke(255);   // Set the color to white
  if (i < 35) {  // When 'i' is less than 35...
    stroke(0);   //...set the color to black
  }
  line(30, i, 80, i);
}
```

Description Allows the program to make a decision about which code to execute. If the **test** evaluates to **true**, the statements enclosed within the block are executed and if the **test** evaluates to **false** the statements are not executed.

Syntax

```
if (test) {
  statements
}
```

Parameters
test any valid expression that evaluates to true or false
statements one or more statements to be executed

Related else

Name

switch

Examples

```
int num = 1;

switch(num) {
  case 0:
    println("Zero");  // Does not execute
    break;
  case 1:
```

```
    println("One");  // Prints "One"
    break;
}
```

```
char letter = 'N';

switch(letter) {
  case 'A':
    println("Alpha");  // Does not execute
    break;
  case 'B':
    println("Bravo");  // Does not execute
    break;
  default:             // Default executes if the case labels
    println("None");   // don't match the switch parameter
    break;
}
```

```
// Removing a "break" enables testing
// for more than one value at once

char letter = 'b';

switch(letter) {
  case 'a':
  case 'A':
    println("Alpha");  // Does not execute
    break;
  case 'b':
  case 'B':
    println("Bravo");  // Prints "Bravo"
    break;
}
```

Description

Works like an **if else** structure, but **switch** is more convenient when you need to select between three or more alternatives. Program controls jumps to the case with the same value as the expression. All remaining statements in the switch are executed unless redirected by a **break**. Only primitive datatypes which can convert to an integer (byte, char, and int) may be used as the **expression** parameter. The default is optional.

Syntax

```
switch(expression)
{
  case label:
     statements
  case label:          // Optional
     statements        // "
  default:             // "
     statements        // "
}
```

Parameters

expression byte, char, or int

label byte, char, or int

statements one or more statements to be executed

Related

case
default
break
if
else

Name

! (logical NOT)

Examples

```
boolean a = false;
if (!a) {
  rect(30, 20, 50, 50);
}
a = true;
if (a) {
  line(20, 10, 90, 80);
  line(20, 80, 90, 10);
}
```

Description Inverts the Boolean value of an expression. Returns **true** if the expression is **false** and returns **false** if the expression is **true**. If the expression **(a>b)** evaluates to true, then **!(a>b)** evaluates to false.

Syntax

```
!expression
```

Parameters **expression** any valid expression

Related
|| (logical OR)
&& (logical AND)
if

Name

&& (logical AND)

Examples

```
for (int i = 5; i <= 95; i += 5) {
  if ((i > 35) && (i < 60)) {
    stroke(0);  // Set color to black
  } else {
    stroke(255);  // Set color to white
  }
  line(30, i, 80, i);
}
```

Description Compares two expressions and returns **true** only if both evaluate to **true**. Returns **false** if one or both evaluate to **false**. The following list shows all possible combinations:

```
true && false // Evaluates false because the second is false
false && true // Evaluates false because the first is false
true && true // Evaluates true because both are true
false && false // Evaluates false because both are false
```

Syntax

```
expression1 && expression2
```

Parameters **expression1** any valid expression
expression2 any valid expression

Related
|| (logical OR)
! (logical NOT)
if

Name

|| (logical OR)

Examples

```
for (int i = 5 ; i <= 95; i += 5) {
  if ((i < 35) || (i > 60)) {
    line(30, i, 80, i);
  }
}
```

Description

Compares two expressions and returns **true** if one or both evaluate to **true**. Returns **false** only if both expressions are **false**. The following list shows all possible combinations:

```
true || false // Evaluates true because the first is true
false || true // Evaluates true because the second is true
true || true // Evaluates true because both are true
false || false // Evaluates false because both are false
```

Syntax

```
expression1 || expression2
```

Parameters

expression1 any valid expression
expression2 any valid expression

Related

&& (logical AND)
! (logical NOT)
if

Name

createShape()

```
PShape square;  // The PShape object

void setup() {
  size(100, 100, P2D);
  // Creating the PShape as a square. The
  // numeric arguments are similar to rect().
  square = createShape(RECT, 0, 0, 80, 80);
}

void draw() {
  shape(square, 10, 10);
}
```

```
PShape s;  // The PShape object
```

Examples

```
void setup() {
  size(100, 100, P2D);
  // Creating a custom PShape as a square, by
  // specifying a series of vertices.
  s = createShape();
  s.beginShape();
  s.fill(0, 0, 255);
  s.noStroke();
  s.vertex(0, 0);
  s.vertex(0, 50);
  s.vertex(50, 50);
  s.vertex(50, 0);
  s.endShape(CLOSE);
}

void draw() {
```

```
  shape(s, 25, 25);
}
```

```
PShape s;

void setup() {
  size(100, 100, P2D);
  s = createShape();
  s.beginShape(TRIANGLE_STRIP);
  s.vertex(30, 75);
  s.vertex(40, 20);
  s.vertex(50, 75);
  s.vertex(60, 20);
  s.vertex(70, 75);
  s.vertex(80, 20);
  s.vertex(90, 75);
  s.endShape();
}

void draw() {
  shape(s, 0, 0);
}
```

```
void setup() {
  size(100, 100, P2D);

  // Create the shape group
  alien = createShape(GROUP);

  // Make two shapes
  head = createShape(ELLIPSE, 0, 25, 50, 50);
  head.setFill(color(255));
  body = createShape(RECT, -25, 45, 50, 40);
  body.setFill(color(0));

  // Add the two "child" shapes to the parent group
  alien.addChild(body);
  alien.addChild(head);
}

void draw() {
  background(204);
  translate(50, 15);
  shape(alien); // Draw the group
}
```

Description

The **createShape()** function is used to define a new shape. Once created, this shape can be drawn with the **shape()** function. The basic way to use the function defines new primitive shapes. One of the following parameters are used as the first parameter: **ELLIPSE**, **RECT**, **ARC**, **TRIANGLE**, **SPHERE**, **BOX**, **QUAD**, **LINE**. The parameters for each of these different shapes are the same as their corrsponding functions: **ellipse()**, **rect()**, **arc()**, **triangle()**, **sphere()**, **box()**, and **line()**. The first example above clarifies how this works.

Custom, unique shapes can be made by using **createShape()** without a parameter. After the shape is started, the drawing attributes and geometry can be set directly to the shape. See the second example above for specifics.

Geometry that groups vertices to build larger forms such as group of triangles can be created with **createShape()** using the same parameters as **beginShape()**.

These options are **POINTS**, **LINES**, **TRIANGLES**, **TRIANGLE_FAN**, **TRIANGLE_STRIP**, **QUADS**, and **QUAD_STRIP**. See the third example above.

The **createShape()** function can also be used to make a complex shape made of other shapes. This is called a "group" and it's created by using the parameter **GROUP** as the first parameter. See the fourth example above to see how it works.

Syntax

```
createShape()
createShape(source)
createShape(type)
createShape(kind, p)
```

Parameters

type	int: either POINTS, LINES, TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP, QUADS, QUAD_STRIP
kind	int: either LINE, TRIANGLE, RECT, ELLIPSE, ARC, SPHERE, BOX
p	float[]: parameters that match the kind of shape

Returns PShape

Related PShape
endShape()
loadShape()

Name

loadShape()

```
PShape s;

void setup() {
  size(100, 100);
  // The file "bot.svg" must be in the data folder
  // of the current sketch to load successfully
  s = loadShape("bot.svg");
}

void draw() {
  shape(s, 10, 10, 80, 80);
}
```

Examples

```
PShape s;

void setup() {
  size(100, 100, P3D);
  // The file "bot.obj" must be in the data folder
  // of the current sketch to load successfully
  s = loadShape("bot.obj");
}

void draw() {
  background(204);
  translate(width/2, height/2);
  shape(s, 0, 0);
}
```

Description Loads geometry into a variable of type **PShape**. SVG and OBJ files may be loaded. To load correctly, the file must be located in the data directory of the current sketch. In most cases, **loadShape()** should be used inside **setup()** because loading shapes inside **draw()** will reduce the speed of a sketch.

Alternatively, the file maybe be loaded from anywhere on the local computer using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows), or the filename parameter can be a URL for a file found on a network.

If the file is not available or an error occurs, **null** will be returned and an error message will be printed to the console. The error message does not halt the program, however the null value may cause a NullPointerException if your code does not check whether the value returned is null.

Syntax
```
loadShape(filename)
loadShape(filename, options)
```

Parameters **filename** String: name of file to load, can be .svg or .obj

Returns PShape

Related PShape
createShape()

Name

PShape

```
PShape s;

void setup() {
  size(100, 100);
  // The file "bot.svg" must be in the data folder
  // of the current sketch to load successfully
  s = loadShape("bot.svg");
}

void draw() {
  shape(s, 10, 10, 80, 80);
}
```

Examples

```
PShape square;  // The PShape object

void setup() {
  size(100, 100, P2D);
  // Creating the PShape as a square. The corner
  // is 0,0 so that the center is at 40,40
  square = createShape(RECT, 0, 0, 80, 80);
}

void draw() {
  shape(square, 10, 10);
}
```

Description Datatype for storing shapes. Before a shape is used, it must be loaded with the **loadShape()** or created with the **createShape()**. The **shape()** function is used to draw the shape to the display window. Processing can currently load and display SVG (Scalable Vector Graphics) and OBJ shapes. OBJ files can only be opened using the **P3D** renderer and **createShape()** is only available with the **P2D** and **P3D** renderers. The **loadShape()** function supports SVG files created with Inkscape and Adobe Illustrator. It is not a full SVG implementation, but offers some straightforward support for handling vector data.

The **PShape** object contains a group of methods that can operate on the shape data. Some of the methods are listed below, but the full list used for creating and

modifying shapes is available here in the Processing Javadoc.

Fields

width	Shape document width
height	Shape document height

Methods

isVisible()	Returns a boolean value "true" if the image is set to be visible, "false" if not
setVisible()	Sets the shape to be visible or invisible
disableStyle()	Disables the shape's style data and uses Processing styles
enableStyle()	Enables the shape's style data and ignores the Processing styles
beginContour()	Starts a new contour
endContour()	Ends a contour
endShape()	Finishes the creation of a new PShape
getChildCount()	Returns the number of children
getChild()	Returns a child element of a shape as a PShape object
addChild()	Adds a new child
getVertexCount()	Returns the total number of vertices as an int
getVertex()	Returns the vertex at the index position
setVertex()	Sets the vertex at the index position
translate()	Displaces the shape
rotateX()	Rotates the shape around the x-axis
rotateY()	Rotates the shape around the y-axis
rotateZ()	Rotates the shape around the z-axis
rotate()	Rotates the shape
scale()	Increases and decreases the size of a shape
resetMatrix()	Replaces the current matrix of a shape with the identity matrix

Constructor `PShape()`

Related

loadShape()
createShape()
shapeMode()

Name

arc()

Examples

```
arc(50, 55, 50, 50, 0, HALF_PI);
noFill();
arc(50, 55, 60, 60, HALF_PI, PI);
arc(50, 55, 70, 70, PI, PI+QUARTER_PI);
arc(50, 55, 80, 80, PI+QUARTER_PI, TWO_PI);
```

```
arc(50, 50, 80, 80, 0, PI+QUARTER_PI, OPEN);
```

```
arc(50, 50, 80, 80, 0, PI+QUARTER_PI, CHORD);
```

```
arc(50, 50, 80, 80, 0, PI+QUARTER_PI, PIE);
```

Description Draws an arc to the screen. Arcs are drawn along the outer edge of an ellipse defined by the **a**, **b**, **c**, and **d** parameters. The origin of the arc's ellipse may be changed with the **ellipseMode()** function. Use the **start** and **stop** parameters to specify the angles (in radians) at which to draw the arc.

There are three ways to draw an arc; the rendering technique used is defined by the optional seventh paramter. The default mode is OPEN, and the other options are CHORD and PIE. Each is shown in the above examples.

Syntax

```
arc(a, b, c, d, start, stop)
arc(a, b, c, d, start, stop, mode)
```

Parameters

- **a** float: x-coordinate of the arc's ellipse
- **b** float: y-coordinate of the arc's ellipse
- **c** float: width of the arc's ellipse by default
- **d** float: height of the arc's ellipse by default
- **start** float: angle to start the arc, specified in radians
- **stop** float: angle to stop the arc, specified in radians

Returns void

Related

ellipse()
ellipseMode()
radians()
degrees()

Name

ellipse()

Examples

```
ellipse(56, 46, 55, 55);
```

Description Draws an ellipse (oval) to the screen. An ellipse with equal width and height is a circle. By default, the first two parameters set the location, and the third and fourth parameters set the shape's width and height. The origin may be changed with the **ellipseMode()** function.

Syntax

```
ellipse(a, b, c, d)
```

Parameters

- **a** float: x-coordinate of the ellipse
- **b** float: y-coordinate of the ellipse
- **c** float: width of the ellipse by default
- **d** float: height of the ellipse by default

Returns void

Related

ellipseMode()
arc()

Name

line()

```
line(30, 20, 85, 75);
```

Examples

```
line(30, 20, 85, 20);
stroke(126);
line(85, 20, 85, 75);
stroke(255);
line(85, 75, 30, 75);
```

```
// Drawing lines in 3D requires P3D
// as a parameter to size()
size(100, 100, P3D);
line(30, 20, 0, 85, 20, 15);
stroke(126);
line(85, 20, 15, 85, 75, 0);
stroke(255);
line(85, 75, 0, 30, 75, -50);
```

Description Draws a line (a direct path between two points) to the screen. The version of **line()** with four parameters draws the line in 2D. To color a line, use the **stroke()** function. A line cannot be filled, therefore the **fill()** function will not affect the color of a line. 2D lines are drawn with a width of one pixel by default, but this can be changed with the **strokeWeight()** function. The version with six parameters allows the line to be placed anywhere within XYZ space. Drawing this shape in 3D with the **z** parameter requires the P3D parameter in combination with **size()** as shown in the above example.

Syntax
```
line(x1, y1, x2, y2)
line(x1, y1, z1, x2, y2, z2)
```

Parameters
x1 float: x-coordinate of the first point
y1 float: y-coordinate of the first point
x2 float: x-coordinate of the second point
y2 float: y-coordinate of the second point
z1 float: z-coordinate of the first point
z2 float: z-coordinate of the second point

Returns void

Related
strokeWeight()
strokeJoin()
strokeCap()
beginShape()

Name

point()

```
noSmooth();
point(30, 20);
point(85, 20);
point(85, 75);
point(30, 75);
```

Examples

```
size(100, 100, P3D);
noSmooth();
point(30, 20, -50);
point(85, 20, -50);
point(85, 75, -50);
point(30, 75, -50);
```

Description Draws a point, a coordinate in space at the dimension of one pixel. The first parameter is the horizontal value for the point, the second value is the vertical value for the point, and the optional third value is the depth value. Drawing this shape in 3D with the **z** parameter requires the P3D parameter in combination with **size()** as shown in the above example.

Syntax
```
point(x, y)
point(x, y, z)
```

xfloat: x-coordinate of the point
Parameters **y**float: y-coordinate of the point
zfloat: z-coordinate of the point
Returns void

Name

quad()

Examples

```
quad(38, 31, 86, 20, 69, 63, 30, 76);
```

Description A quad is a quadrilateral, a four sided polygon. It is similar to a rectangle, but the angles between its edges are not constrained to ninety degrees. The first pair of parameters (x1,y1) sets the first vertex and the subsequent pairs should proceed clockwise or counter-clockwise around the defined shape.

Syntax

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

Parameters
x1float: x-coordinate of the first corner
y1float: y-coordinate of the first corner
x2float: x-coordinate of the second corner
y2float: y-coordinate of the second corner
x3float: x-coordinate of the third corner
y3float: y-coordinate of the third corner
x4float: x-coordinate of the fourth corner
y4float: y-coordinate of the fourth corner

Returns void

Name

rect()

Examples

```
rect(30, 20, 55, 55);
```

```
rect(30, 20, 55, 55, 7);
```

```
rect(30, 20, 55, 55, 3, 6, 12, 18);
```

Description Draws a rectangle to the screen. A rectangle is a four-sided shape with every angle at ninety degrees. By default, the first two parameters set the location of the upper-left corner, the third sets the width, and the fourth sets the height. The way these parameters are interpreted, however, may be changed with the **rectMode()** function.

To draw a rounded rectangle, add a fifth parameter, which is used as the radius value for all four corners.

To use a different radius value for each corner, include eight parameters. When using eight parameters, the latter four set the radius of the arc at each corner separately, starting with the top-left corner and moving clockwise around the rectangle.

Syntax

```
rect(a, b, c, d)
rect(a, b, c, d, r)
rect(a, b, c, d, tl, tr, br, bl)
```

Parameters

a float: x-coordinate of the rectangle by default
b float: y-coordinate of the rectangle by default
c float: width of the rectangle by default
d float: height of the rectangle by default
r float: radii for all four corners
tl float: radius for top-left corner
tr float: radius for top-right corner
br float: radius for bottom-right corner
bl float: radius for bottom-left corner

Returns void

Related rectMode()
quad()

Name

triangle()

Examples

```
triangle(30, 75, 58, 20, 86, 75);
```

Description A triangle is a plane created by connecting three points. The first two arguments specify the first point, the middle two arguments specify the second point, and the last two arguments specify the third point.

Syntax

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

Parameters

x1 float: x-coordinate of the first point
y1 float: y-coordinate of the first point
x2 float: x-coordinate of the second point
y2 float: y-coordinate of the second point
x3 float: x-coordinate of the third point
y3 float: y-coordinate of the third point

Returns void

Related beginShape()

Name

bezier()

Examples

```
noFill();
stroke(255, 102, 0);
line(85, 20, 10, 10);
line(90, 90, 15, 80);
stroke(0, 0, 0);
bezier(85, 20, 10, 10, 90, 90, 15, 80);
```

```
noFill();
stroke(255, 102, 0);
line(30, 20, 80, 5);
line(80, 75, 30, 75);
stroke(0, 0, 0);
```

```
bezier(30, 20,  80, 5,  80, 75,  30, 75);
```

Description Draws a Bezier curve on the screen. These curves are defined by a series of anchor and control points. The first two parameters specify the first anchor point and the last two parameters specify the other anchor point. The middle parameters specify the control points which define the shape of the curve. Bezier curves were developed by French engineer Pierre Bezier. Using the 3D version requires rendering with P3D (see the Environment reference for more information).

Syntax
```
bezier(x1, y1, x2, y2, x3, y3, x4, y4)
bezier(x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4)
```

Parameters
x1 float: coordinates for the first anchor point
y1 float: coordinates for the first anchor point
z1 float: coordinates for the first anchor point
x2 float: coordinates for the first control point
y2 float: coordinates for the first control point
z2 float: coordinates for the first control point
x3 float: coordinates for the second control point
y3 float: coordinates for the second control point
z3 float: coordinates for the second control point
x4 float: coordinates for the second anchor point
y4 float: coordinates for the second anchor point
z4 float: coordinates for the second anchor point

Returns void

Related bezierVertex()
curve()

Name

bezierDetail()

```
// Move the mouse left and right to see the detail change

void setup() {
  size(100, 100, P3D);
  noFill();
}
```

Examples

```
void draw() {
  background(204);
  int d = int(map(mouseX, 0, 100, 1, 20));
  bezierDetail(d);
  bezier(85, 20, 10, 10, 90, 90, 15, 80);
}
```

Description Sets the resolution at which Beziers display. The default value is 20. This function is only useful when using the **P3D** renderer; the default **P2D** renderer does not use this information.

Syntax
```
bezierDetail(detail)
```

Parameters **detail** int: resolution of the curves

Returns void

Related curve()
curveVertex()
curveTightness()

Name

bezierPoint()

Examples

```
noFill();
bezier(85, 20, 10, 10, 90, 90, 15, 80);
fill(255);
int steps = 10;
for (int i = 0; i <= steps; i++) {
  float t = i / float(steps);
  float x = bezierPoint(85, 10, 90, 15, t);
  float y = bezierPoint(20, 10, 90, 80, t);
  ellipse(x, y, 5, 5);
}
```

Description Evaluates the Bezier at point t for points a, b, c, d. The parameter t varies between 0 and 1, a and d are points on the curve, and b and c are the control points. This can be done once with the x coordinates and a second time with the y coordinates to get the location of a bezier curve at t.

Syntax

```
bezierPoint(a, b, c, d, t)
```

Parameters

a float: coordinate of first point on the curve
b float: coordinate of first control point
c float: coordinate of second control point
d float: coordinate of second point on the curve
t float: value between 0 and 1

Returns float

Related bezier()
bezierVertex()
curvePoint()

Name

bezierTangent()

Examples

```
noFill();
bezier(85, 20, 10, 10, 90, 90, 15, 80);
int steps = 6;
fill(255);
for (int i = 0; i <= steps; i++) {
  float t = i / float(steps);
  // Get the location of the point
  float x = bezierPoint(85, 10, 90, 15, t);
  float y = bezierPoint(20, 10, 90, 80, t);
  // Get the tangent points
  float tx = bezierTangent(85, 10, 90, 15, t);
  float ty = bezierTangent(20, 10, 90, 80, t);
  // Calculate an angle from the tangent points
  float a = atan2(ty, tx);
  a += PI;
  stroke(255, 102, 0);
  line(x, y, cos(a)*30 + x, sin(a)*30 + y);
  // The following line of code makes a line
  // inverse of the above line
  //line(x, y, cos(a)*-30 + x, sin(a)*-30 + y);
```

```
  stroke(0);
  ellipse(x, y, 5, 5);
}
```

```
noFill();
bezier(85, 20, 10, 10, 90, 90, 15, 80);
stroke(255, 102, 0);
int steps = 16;
for (int i = 0; i <= steps; i++) {
  float t = i / float(steps);
  float x = bezierPoint(85, 10, 90, 15, t);
  float y = bezierPoint(20, 10, 90, 80, t);
  float tx = bezierTangent(85, 10, 90, 15, t);
  float ty = bezierTangent(20, 10, 90, 80, t);
  float a = atan2(ty, tx);
  a -= HALF_PI;
  line(x, y, cos(a)*8 + x, sin(a)*8 + y);
}
```

Description Calculates the tangent of a point on a Bezier curve. There is a good definition of *tangent* on Wikipedia.

Syntax
```
bezierTangent(a, b, c, d, t)
```

Parameters
- **a** float: coordinate of first point on the curve
- **b** float: coordinate of first control point
- **c** float: coordinate of second control point
- **d** float: coordinate of second point on the curve
- **t** float: value between 0 and 1

Returns float

Related bezier()
bezierVertex()
curvePoint()

Name

curve()

Examples
```
noFill();
stroke(255, 102, 0);
curve(5, 26, 5, 26, 73, 24, 73, 61);
stroke(0);
curve(5, 26, 73, 24, 73, 61, 15, 65);
stroke(255, 102, 0);
curve(73, 24, 73, 61, 15, 65, 15, 65);
```

Description Draws a curved line on the screen. The first and second parameters specify the beginning control point and the last two parameters specify the ending control point. The middle parameters specify the start and stop of the curve. Longer curves can be created by putting a series of **curve()** functions together or using **curveVertex()**. An additional function called **curveTightness()** provides control for the visual quality of the curve. The **curve()** function is an implementation of Catmull-Rom splines. Using the 3D version requires rendering with P3D (see the Environment reference for more information).

Syntax
```
curve(x1, y1, x2, y2, x3, y3, x4, y4)
curve(x1, y1, z1, x2, y2, z2, x3, y3, z3, x4, y4, z4)
```

Parameters
- **x1** float: coordinates for the beginning control point

y1float: coordinates for the beginning control point
x2float: coordinates for the first point
y2float: coordinates for the first point
x3float: coordinates for the second point
y3float: coordinates for the second point
x4float: coordinates for the ending control point
y4float: coordinates for the ending control point
z1float: coordinates for the beginning control point
z2float: coordinates for the first point
z3float: coordinates for the second point
z4float: coordinates for the ending control point

Returns void

Related curveVertex()
curveTightness()
bezier()

Name

curveDetail()

Examples

```
void setup() {
  size(100, 100, P3D);
  noFill();
  noLoop();
}

void draw() {
  curveDetail(1);
  drawCurves(-15);
  stroke(126);
  curveDetail(2);
  drawCurves(0);
  stroke(255);
  curveDetail(4);
  drawCurves(15);
}

void drawCurves(float y) {
  curve( 5, 28+y,  5, 28+y, 73, 26+y, 73, 63+y);
  curve( 5, 28+y, 73, 26+y, 73, 63+y, 15, 67+y);
  curve(73, 26+y, 73, 63+y, 15, 67+y, 15, 67+y);
}
```

Description Sets the resolution at which curves display. The default value is 20. This function is only useful when using the P3D renderer as the default P2D renderer does not use this information.

Syntax `curveDetail(detail)`

Parameters **detail**int: resolution of the curves

Returns void

Related curve()
curveVertex()
curveTightness()

Name

curvePoint()

Examples

```
noFill();
curve(5, 26, 5, 26, 73, 24, 73, 61);
curve(5, 26, 73, 24, 73, 61, 15, 65);
fill(255);
ellipseMode(CENTER);
int steps = 6;
for (int i = 0; i <= steps; i++) {
  float t = i / float(steps);
  float x = curvePoint(5, 5, 73, 73, t);
  float y = curvePoint(26, 26, 24, 61, t);
  ellipse(x, y, 5, 5);
  x = curvePoint(5, 73, 73, 15, t);
  y = curvePoint(26, 24, 61, 65, t);
  ellipse(x, y, 5, 5);
}
```

Description Evaluates the curve at point **t** for points **a**, **b**, **c**, **d**. The parameter **t** may range from 0 (the start of the curve) and 1 (the end of the curve). **a** and **d** are points on the curve, and **b** and **c** are the control points. This can be used once with the **x** coordinates and a second time with the **y** coordinates to get the location of a curve at **t**.

Syntax `curvePoint(a, b, c, d, t)`

Parameters
- **a** float: coordinate of first point on the curve
- **b** float: coordinate of second point on the curve
- **c** float: coordinate of third point on the curve
- **d** float: coordinate of fourth point on the curve
- **t** float: value between 0 and 1

Returns float

Related curve()
curveVertex()
bezierPoint()

Name

curveTangent()

Examples

```
noFill();
curve(5, 26, 73, 24, 73, 61, 15, 65);
int steps = 6;
for (int i = 0; i <= steps; i++) {
  float t = i / float(steps);
  float x = curvePoint(5, 73, 73, 15, t);
  float y = curvePoint(26, 24, 61, 65, t);
  //ellipse(x, y, 5, 5);
  float tx = curveTangent(5, 73, 73, 15, t);
  float ty = curveTangent(26, 24, 61, 65, t);
  float a = atan2(ty, tx);
  a -= PI/2.0;
  line(x, y, cos(a)*8 + x, sin(a)*8 + y);
}
```

Description Calculates the tangent of a point on a curve. There's a good definition of *tangent* on Wikipedia.

Syntax
```
curveTangent(a, b, c, d, t)
```

Parameters
- **a** float: coordinate of first point on the curve
- **b** float: coordinate of first control point
- **c** float: coordinate of second control point
- **d** float: coordinate of second point on the curve
- **t** float: value between 0 and 1

Returns float

Related
curve()
curveVertex()
curvePoint()
bezierTangent()

Name

curveTightness()

Examples

```
// Move the mouse left and right to see the curve change

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

void draw() {
  background(204);
  float t = map(mouseX, 0, width, -5, 5);
  curveTightness(t);
  beginShape();
  curveVertex(10, 26);
  curveVertex(10, 26);
  curveVertex(83, 24);
  curveVertex(83, 61);
  curveVertex(25, 65);
  curveVertex(25, 65);
  endShape();
}
```

Description Modifies the quality of forms created with **curve()** and **curveVertex()**. The parameter **tightness** determines how the curve fits to the vertex points. The value 0.0 is the default value for **tightness** (this value defines the curves to be Catmull-Rom splines) and the value 1.0 connects all the points with straight lines. Values within the range -5.0 and 5.0 will deform the curves but will leave them recognizable and as values increase in magnitude, they will continue to deform.

Syntax
```
curveTightness(tightness)
```

Parameters
- **tightness** float: amount of deformation from the original vertices

Returns void

Related
curve()
curveVertex()

Name

box()

Examples

```
size(100, 100, P3D);
translate(58, 48, 0);
rotateY(0.5);
noFill();
box(40);
```

```
size(100, 100, P3D);
translate(58, 48, 0);
rotateY(0.5);
noFill();
box(40, 20, 50);
```

Description A box is an extruded rectangle. A box with equal dimensions on all sides is a cube.

Syntax

```
box(size)
box(w, h, d)
```

Parameters
size float: dimension of the box in all dimensions (creates a cube)
w float: dimension of the box in the x-dimension
h float: dimension of the box in the y-dimension
d float: dimension of the box in the z-dimension

Returns void

Related sphere()

Name

sphere()

Examples

```
noStroke();
lights();
translate(58, 48, 0);
sphere(28);
```

Description A sphere is a hollow ball made from tessellated triangles.

Syntax

```
sphere(r)
```

Parameters **r** float: the radius of the sphere

Returns void

Related sphereDetail()

Name

sphereDetail()

Examples

```
void setup() {
  size(100, 100, P3D);
}

void draw() {
  background(200);
  stroke(255, 50);
  translate(50, 50, 0);
  rotateX(mouseY * 0.05);
  rotateY(mouseX * 0.05);
  fill(mouseX * 2, 0, 160);
  sphereDetail(mouseX / 4);
```

```
  sphere(40);
}
```

Description

Controls the detail used to render a sphere by adjusting the number of vertices of the sphere mesh. The default resolution is 30, which creates a fairly detailed sphere definition with vertices every 360/30 = 12 degrees. If you're going to render a great number of spheres per frame, it is advised to reduce the level of detail using this function. The setting stays active until **sphereDetail()** is called again with a new parameter and so should *not* be called prior to every **sphere()** statement, unless you wish to render spheres with different settings, e.g. using less detail for smaller spheres or ones further away from the camera. To control the detail of the horizontal and vertical resolution independently, use the version of the functions with two parameters.

Syntax

```
sphereDetail(res)
sphereDetail(ures, vres)
```

Parameters

res int: number of segments (minimum 3) used per full circle revolution
ures int: number of segments used longitudinally per full circle revolutoin
vres int: number of segments used latitudinally from top to bottom

Returns void

Related sphere()

Name

ellipseMode()

Examples

```
ellipseMode(RADIUS);  // Set ellipseMode to RADIUS
fill(255);  // Set fill to white
ellipse(50, 50, 30, 30);  // Draw white ellipse using RADIUS mode

ellipseMode(CENTER);  // Set ellipseMode to CENTER
fill(100);  // Set fill to gray
ellipse(50, 50, 30, 30);  // Draw gray ellipse using CENTER mode
```

```
ellipseMode(CORNER);  // Set ellipseMode is CORNER
fill(255);  // Set fill to white
ellipse(25, 25, 50, 50);  // Draw white ellipse using CORNER mode

ellipseMode(CORNERS);  // Set ellipseMode to CORNERS
fill(100);  // Set fill to gray
ellipse(25, 25, 50, 50);  // Draw gray ellipse using CORNERS mode
```

Description

Modifies the location from which ellipses are drawn by changing the way in which parameters given to **ellipse()** are intepreted.

The default mode is **ellipseMode(CENTER)**, which interprets the first two parameters of **ellipse()** as the shape's center point, while the third and fourth parameters are its width and height.

ellipseMode(RADIUS) also uses the first two parameters of **ellipse()** as the shape's center point, but uses the third and fourth parameters to specify half of the shapes's width and height.

ellipseMode(CORNER) interprets the first two parameters of **ellipse()** as the upper-left corner of the shape, while the third and fourth parameters are its width

and height.

ellipseMode(CORNERS) interprets the first two parameters of **ellipse()** as the location of one corner of the ellipse's bounding box, and the third and fourth parameters as the location of the opposite corner.

The parameter must be written in ALL CAPS because Processing is a case-sensitive language.

Syntax `ellipseMode(mode)`

Parameters **mode** int: either CENTER, RADIUS, CORNER, or CORNERS

Returns void

Related ellipse()
arc()

Name

noSmooth()

Examples

```
background(0);
noStroke();
smooth();
ellipse(30, 48, 36, 36);
noSmooth();
ellipse(70, 48, 36, 36);
```

Description Draws all geometry with jagged (aliased) edges. Note that **smooth()** is active by default, so it is necessary to call **noSmooth()** to disable smoothing of geometry, images, and fonts.

Syntax `noSmooth()`

Returns void

Related smooth()

Name

rectMode()

Examples

```
rectMode(CORNER);  // Default rectMode is CORNER
fill(255);  // Set fill to white
rect(25, 25, 50, 50);  // Draw white rect using CORNER mode

rectMode(CORNERS);  // Set rectMode to CORNERS
fill(100);  // Set fill to gray
rect(25, 25, 50, 50);  // Draw gray rect using CORNERS mode

rectMode(RADIUS);  // Set rectMode to RADIUS
fill(255);  // Set fill to white
rect(50, 50, 30, 30);  // Draw white rect using RADIUS mode

rectMode(CENTER);  // Set rectMode to CENTER
fill(100);  // Set fill to gray
rect(50, 50, 30, 30);  // Draw gray rect using CENTER mode
```

Description Modifies the location from which rectangles are drawn by changing the way in

which parameters given to **rect()** are intepreted.

The default mode is **rectMode(CORNER)**, which interprets the first two parameters of **rect()** as the upper-left corner of the shape, while the third and fourth parameters are its width and height.

rectMode(CORNERS) interprets the first two parameters of **rect()** as the location of one corner, and the third and fourth parameters as the location of the opposite corner.

rectMode(CENTER) interprets the first two parameters of **rect()** as the shape's center point, while the third and fourth parameters are its width and height.

rectMode(RADIUS) also uses the first two parameters of **rect()** as the shape's center point, but uses the third and fourth parameters to specify half of the shapes's width and height.

The parameter must be written in ALL CAPS because Processing is a case-sensitive language.

Syntax `rectMode(mode)`

Parameters **mode** int: either CORNER, CORNERS, CENTER, or RADIUS

Returns void

Related rect()

Name

smooth()

Examples

```
background(0);
noStroke();
smooth();
ellipse(30, 48, 36, 36);
noSmooth();
ellipse(70, 48, 36, 36);
```

Description Draws all geometry with smooth (anti-aliased) edges. **smooth()** will also improve image quality of resized images. Note that **smooth()** is active by default; **noSmooth()** can be used to disable smoothing of geometry, images, and fonts.

The **level** parameter increases the level of smoothness with the P2D and P3D renderers. This is the level of over sampling applied to the graphics buffer. The value "2" will double the rendering size before scaling it down to the display size. This is called "2x anti-aliasing." The value 4 is used for 4x anti-aliasing and 8 is specified for 8x anti-aliasing. If **level** is set to 0, it will disable all smoothing; it's the equivalent of the function **noSmooth()**. The maximum anti-aliasing level is determined by the hardware of the machine that is running the software.

With the default renderer, **smooth(2)** is bilinear and **smooth(4)** is bicubic. Nothing implemented on Android 2D.

Syntax

```
smooth()
smooth(level)
```

Parameters **level** int: either 2, 4, or 8

Returns void

Related noSmooth()
size()

Name

strokeCap()

Examples

```
strokeWeight(12.0);
strokeCap(ROUND);
line(20, 30, 80, 30);
strokeCap(SQUARE);
line(20, 50, 80, 50);
strokeCap(PROJECT);
line(20, 70, 80, 70);
```

Description Sets the style for rendering line endings. These ends are either squared, extended, or rounded, each of which specified with the corresponding parameters: SQUARE, PROJECT, and ROUND. The default cap is ROUND.

Syntax

```
strokeCap(cap)
```

Parameters **cap** int: either SQUARE, PROJECT, or ROUND

Returns void

Related stroke()
strokeWeight()
strokeJoin()
size()

Name

strokeJoin()

```
noFill();
strokeWeight(10.0);
strokeJoin(MITER);
beginShape();
vertex(35, 20);
vertex(65, 50);
vertex(35, 80);
endShape();
```

Examples

```
noFill();
strokeWeight(10.0);
strokeJoin(BEVEL);
beginShape();
vertex(35, 20);
vertex(65, 50);
vertex(35, 80);
endShape();
```

```
noFill();
strokeWeight(10.0);
strokeJoin(ROUND);
```

```
beginShape();
vertex(35, 20);
vertex(65, 50);
vertex(35, 80);
endShape();
```

Description Sets the style of the joints which connect line segments. These joints are either mitered, beveled, or rounded and specified with the corresponding parameters MITER, BEVEL, and ROUND. The default joint is MITER.

Syntax `strokeJoin(join)`

Parameters **join** int: either MITER, BEVEL, ROUND

Returns void

Related stroke()
strokeWeight()
strokeCap()

Name

strokeWeight()

Examples

```
strokeWeight(1);  // Default
line(20, 20, 80, 20);
strokeWeight(4);  // Thicker
line(20, 40, 80, 40);
strokeWeight(10);  // Beastly
line(20, 70, 80, 70);
```

Description Sets the width of the stroke used for lines, points, and the border around shapes. All widths are set in units of pixels.

Syntax `strokeWeight(weight)`

Parameters **weight** float: the weight (in pixels) of the stroke

Returns void

Related stroke()
strokeJoin()
strokeCap()

Name

beginContour()

Examples

```
size(100, 100, P2D);
translate(50, 50);
stroke(255, 0, 0);
beginShape();
// Exterior part of shape
vertex(-40, -40);
vertex(40, -40);
vertex(40, 40);
vertex(-40, 40);
// Interior part of shape
beginContour();
vertex(-20, -20);
vertex(20, -20);
vertex(20, 20);
```

```
vertex(-20, 20);
endContour();
endShape(CLOSE);
```

Use the **beginContour()** and **endContour()** function to create negative shapes within shapes. For instance, the center of the letter 'O'. **beginContour()** begins recording vertices for the shape and **endContour()** stops recording. These functions can only be within a **beginShape()**/**endShape()** pair and they only work with the P2D and P3D renderers.

Description

Transformations such as **translate()**, **rotate()**, and **scale()** do not work within a **beginContour()**/**endContour()** pair. It is also not possible to use other shapes, such as **ellipse()** or **rect()** within.

Syntax `beginContour()`

Returns void

Name

beginShape()

```
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape(CLOSE);
```

```
beginShape(POINTS);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

Examples

```
beginShape(LINES);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

```
noFill();
beginShape();
vertex(30, 20);
vertex(85, 20);
```

```
vertex(85, 75);
vertex(30, 75);
endShape(CLOSE);


beginShape(TRIANGLES);
vertex(30, 75);
vertex(40, 20);
vertex(50, 75);
vertex(60, 20);
vertex(70, 75);
vertex(80, 20);
endShape();


beginShape(TRIANGLE_STRIP);
vertex(30, 75);
vertex(40, 20);
vertex(50, 75);
vertex(60, 20);
vertex(70, 75);
vertex(80, 20);
vertex(90, 75);
endShape();


beginShape(TRIANGLE_FAN);
vertex(57.5, 50);
vertex(57.5, 15);
vertex(92, 50);
vertex(57.5, 85);
vertex(22, 50);
vertex(57.5, 15);
endShape();


beginShape(QUADS);
vertex(30, 20);
vertex(30, 75);
vertex(50, 75);
vertex(50, 20);
vertex(65, 20);
vertex(65, 75);
vertex(85, 75);
vertex(85, 20);
endShape();


beginShape(QUAD_STRIP);
vertex(30, 20);
vertex(30, 75);
vertex(50, 20);
vertex(50, 75);
vertex(65, 20);
vertex(65, 75);
vertex(85, 20);
vertex(85, 75);
endShape();


beginShape();
```

```
vertex(20, 20);
vertex(40, 20);
vertex(40, 40);
vertex(60, 40);
vertex(60, 60);
vertex(20, 60);
endShape(CLOSE);
```

Description Using the **beginShape()** and **endShape()** functions allow creating more complex forms. **beginShape()** begins recording vertices for a shape and **endShape()** stops recording. The value of the **kind** parameter tells it which types of shapes to create from the provided vertices. With no mode specified, the shape can be any irregular polygon. The parameters available for beginShape() are POINTS, LINES, TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP, QUADS, and QUAD_STRIP. After calling the **beginShape()** function, a series of **vertex()** commands must follow. To stop drawing the shape, call **endShape()**. The **vertex()** function with two parameters specifies a position in 2D and the **vertex()** function with three parameters specifies a position in 3D. Each shape will be outlined with the current stroke color and filled with the fill color.

Transformations such as **translate()**, **rotate()**, and **scale()** do not work within **beginShape()**. It is also not possible to use other shapes, such as **ellipse()** or **rect()** within **beginShape()**.

The P3D renderer settings allow **stroke()** and **fill()** settings to be altered per-vertex, but P2D and the default renderer do not. Settings such as **strokeWeight()**, **strokeCap()**, and **strokeJoin()** cannot be changed while inside a **beginShape()/endShape()** block with any renderer.

Syntax
```
beginShape()
beginShape(kind)
```

Parameters **kind** int: Either POINTS, LINES, TRIANGLES, TRIANGLE_FAN, TRIANGLE_STRIP, QUADS, or QUAD_STRIP

Returns void

Related
PShape
endShape()
vertex()
curveVertex()
bezierVertex()

Name

bezierVertex()

Examples
```
noFill();
beginShape();
vertex(30, 20);
bezierVertex(80, 0, 80, 75, 30, 75);
endShape();
```

```
beginShape();
vertex(30, 20);
bezierVertex(80, 0, 80, 75, 30, 75);
bezierVertex(50, 80, 60, 25, 30, 20);
```

```
endShape();
```

Description Specifies vertex coordinates for Bezier curves. Each call to **bezierVertex()** defines the position of two control points and one anchor point of a Bezier curve, adding a new segment to a line or shape. The first time **bezierVertex()** is used within a **beginShape()** call, it must be prefaced with a call to **vertex()** to set the first anchor point. This function must be used between **beginShape()** and **endShape()** and only when there is no MODE parameter specified to **beginShape()**. Using the 3D version requires rendering with P3D (see the Environment reference for more information).

Syntax
```
bezierVertex(x2, y2, x3, y3, x4, y4)
bezierVertex(x2, y2, z2, x3, y3, z3, x4, y4, z4)
```

Parameters
x2 float: the x-coordinate of the 1st control point
y2 float: the y-coordinate of the 1st control point
z2 float: the z-coordinate of the 1st control point
x3 float: the x-coordinate of the 2nd control point
y3 float: the y-coordinate of the 2nd control point
z3 float: the z-coordinate of the 2nd control point
x4 float: the x-coordinate of the anchor point
y4 float: the y-coordinate of the anchor point
z4 float: the z-coordinate of the anchor point

Returns void

Related curveVertex()
vertex()
quadraticVertex()
bezier()

Name

curveVertex()

Examples
```
noFill();
beginShape();
curveVertex(84,  91);
curveVertex(84,  91);
curveVertex(68,  19);
curveVertex(21,  17);
curveVertex(32, 100);
curveVertex(32, 100);
endShape();
```

Description Specifies vertex coordinates for curves. This function may only be used between **beginShape()** and **endShape()** and only when there is no MODE parameter specified to **beginShape()**. The first and last points in a series of **curveVertex()** lines will be used to guide the beginning and end of a the curve. A minimum of four points is required to draw a tiny curve between the second and third points. Adding a fifth point with **curveVertex()** will draw the curve between the second, third, and fourth points. The **curveVertex()** function is an implementation of Catmull-Rom splines. Using the 3D version requires rendering with P3D (see the Environment reference for more information).

Syntax
```
curveVertex(x, y)
curveVertex(x, y, z)
```

Parameters
x float: the x-coordinate of the vertex
y float: the y-coordinate of the vertex

zfloat: the z-coordinate of the vertex

Returns void

Related
curve()
beginShape()
endShape()
vertex()
bezier()
quadraticVertex()

Name

endContour()

Examples

```
size(100, 100, P2D);
translate(50, 50);
stroke(255, 0, 0);
beginShape();
// Exterior part of shape
vertex(-40, -40);
vertex(40, -40);
vertex(40, 40);
vertex(-40, 40);
// Interior part of shape
beginContour();
vertex(-20, -20);
vertex(20, -20);
vertex(20, 20);
vertex(-20, 20);
endContour();
endShape(CLOSE);
```

Description Use the **beginContour()** and **endContour()** function to create negative shapes within shapes. For instance, the center of the letter 'O'. **beginContour()** begins recording vertices for the shape and **endContour()** stops recording. These functions can only be within a **beginShape()**/**endShape()** pair and they only work with the P2D and P3D renderers.

Transformations such as **translate()**, **rotate()**, and **scale()** do not work within a **beginContour()**/**endContour()** pair. It is also not possible to use other shapes, such as **ellipse()** or **rect()** within.

Syntax `endContour()`

Returns void

Name

endShape()

Examples

```
noFill();

beginShape();
vertex(20, 20);
vertex(45, 20);
vertex(45, 80);
```

```
endShape(CLOSE);

beginShape();
vertex(50, 20);
vertex(75, 20);
vertex(75, 80);
endShape();
```

Description The **endShape()** function is the companion to **beginShape()** and may only be called after **beginShape()**. When **endshape()** is called, all of image data defined since the previous call to **beginShape()** is written into the image buffer. The constant CLOSE as the value for the MODE parameter to close the shape (to connect the beginning and the end).

Syntax
```
endShape()
endShape(mode)
```

Parameters **mode** int: use CLOSE to close the shape

Returns void

Related PShape
beginShape()

Name

quadraticVertex()

```
noFill();
strokeWeight(4);
beginShape();
vertex(20, 20);
quadraticVertex(80, 20, 50, 50);
endShape();
```

Examples

```
noFill();
strokeWeight(4);
beginShape();
vertex(20, 20);
quadraticVertex(80, 20, 50, 50);
quadraticVertex(20, 80, 80, 80);
vertex(80, 60);
endShape();
```

Description Specifies vertex coordinates for quadratic Bezier curves. Each call to **quadraticVertex()** defines the position of one control points and one anchor point of a Bezier curve, adding a new segment to a line or shape. The first time **quadraticVertex()** is used within a **beginShape()** call, it must be prefaced with a call to **vertex()** to set the first anchor point. This function must be used between **beginShape()** and **endShape()** and only when there is no MODE parameter specified to **beginShape()**. Using the 3D version requires rendering with P3D (see the Environment reference for more information).

Syntax
```
quadraticVertex(cx, cy, x3, y3)
quadraticVertex(cx, cy, cz, x3, y3, z3)
```

Parameters
cx float: the x-coordinate of the control point
cy float: the y-coordinate of the control point
x3 float: the x-coordinate of the anchor point
y3 float: the y-coordinate of the anchor point
cz float: the z-coordinate of the control point

z3float: the z-coordinate of the anchor point

Returns void

Related curveVertex()
vertex()
bezierVertex()
bezier()

Name

vertex()

Examples

```
beginShape(POINTS);
vertex(30, 20);
vertex(85, 20);
vertex(85, 75);
vertex(30, 75);
endShape();
```

```
// Drawing vertices in 3D requires P3D
// as a parameter to size()
size(100, 100, P3D);
beginShape(POINTS);
vertex(30, 20, -50);
vertex(85, 20, -50);
vertex(85, 75, -50);
vertex(30, 75, -50);
endShape();
```

```
size(100, 100, P3D);
PImage img = loadImage("laDefense.jpg");
noStroke();
beginShape();
texture(img);
// "laDefense.jpg" is 100x100 pixels in size so
// the values 0 and 100 are used for the
// parameters "u" and "v" to map it directly
// to the vertex points
vertex(10, 20, 0, 0);
vertex(80, 5, 100, 0);
vertex(95, 90, 100, 100);
vertex(40, 95, 0, 100);
endShape();
```

Description

All shapes are constructed by connecting a series of vertices. **vertex()** is used to specify the vertex coordinates for points, lines, triangles, quads, and polygons. It is used exclusively within the **beginShape()** and **endShape()** functions.

Drawing a vertex in 3D using the **z** parameter requires the P3D parameter in combination with size, as shown in the above example.

This function is also used to map a texture onto geometry. The **texture()** function declares the texture to apply to the geometry and the **u** and **v** coordinates set define the mapping of this texture to the form. By default, the coordinates used for **u** and **v** are specified in relation to the image's size in pixels, but this relation

can be changed with **textureMode()**.

Syntax

```
vertex(x, y)
vertex(x, y, z)
vertex(v)
vertex(x, y, u, v)
vertex(x, y, z, u, v)
```

Parameters

v float[]: vertex parameters, as a float array of length VERTEX_FIELD_COUNT
x float: x-coordinate of the vertex
y float: y-coordinate of the vertex
z float: z-coordinate of the vertex
u float: horizontal coordinate for the texture mapping
v float: vertical coordinate for the texture mapping

Returns void

Related

beginShape()
endShape()
bezierVertex()
quadraticVertex()
curveVertex()
texture()

Name

shape()

Examples

```
PShape s;

void setup() {
  s = loadShape("bot.svg");
}

void draw() {
  shape(s, 10, 10, 80, 80);
}
```

Description

Draws shapes to the display window. Shapes must be in the sketch's "data" directory to load correctly. Select "Add file..." from the "Sketch" menu to add the shape. Processing currently works with SVG, OBJ, and custom-created shapes. The **shape** parameter specifies the shape to display and the coordinate parameters define the location of the shape from its upper-left corner. The shape is displayed at its original size unless the **c** and **d** parameters specify a different size. The **shapeMode()** function can be used to change the way these parameters are interpreted.

Syntax

```
shape(shape)
shape(shape, x, y)
shape(shape, a, b, c, d)
```

Parameters

shape PShape: the shape to display
x float: x-coordinate of the shape
y float: y-coordinate of the shape
a float: x-coordinate of the shape
b float: y-coordinate of the shape
c float: width to display the shape
d float: height to display the shape

Returns void

PShape

Related loadShape()

shapeMode()

Name

shapeMode()

Examples

```
PShape bot

void setup() {
  size(100, 100);
  bot = loadShape("bot.svg");
}

void draw() {
  shapeMode(CENTER);
  shape(bot, 35, 35, 50, 50);
  shapeMode(CORNER);
  shape(bot, 35, 35, 50, 50);
}
```

Description Modifies the location from which shapes draw. The default mode is **shapeMode(CORNER)**, which specifies the location to be the upper left corner of the shape and uses the third and fourth parameters of **shape()** to specify the width and height. The syntax **shapeMode(CORNERS)** uses the first and second parameters of **shape()** to set the location of one corner and uses the third and fourth parameters to set the opposite corner. The syntax **shapeMode(CENTER)** draws the shape from its center point and uses the third and forth parameters of **shape()** to specify the width and height. The parameter must be written in "ALL CAPS" because Processing is a case sensitive language.

Syntax `shapeMode(mode)`

Parameters **mode** int: either CORNER, CORNERS, CENTER

Returns void

PShape

Related shape()

rectMode()

Name

mouseButton

Examples

```
// Click within the image and press
// the left and right mouse buttons to
// change the value of the rectangle
void draw() {
  if (mousePressed && (mouseButton == LEFT)) {
    fill(0);
  } else if (mousePressed && (mouseButton == RIGHT)) {
    fill(255);
  } else {
    fill(126);
  }
  rect(25, 25, 50, 50);
```

```
}
```

```
// Click within the image and press
// the left and right mouse buttons to
// change the value of the rectangle
void draw() {
  rect(25, 25, 50, 50);
}

void mousePressed() {
  if (mouseButton == LEFT) {
    fill(0);
  } else if (mouseButton == RIGHT) {
    fill(255);
  } else {
    fill(126);
  }
}
```

Description Processing automatically tracks if the mouse button is pressed and which button is pressed. The value of the system variable **mouseButton** is either **LEFT**, **RIGHT**, or **CENTER** depending on which button is pressed.

Related
mouseX
mouseY
mousePressed
mouseReleased()
mouseMoved()
mouseDragged()

Name

mouseClicked()

Examples

```
// Click within the image to change
// the value of the rectangle after
// after the mouse has been clicked

int value = 0;

void draw() {
  fill(value);
  rect(25, 25, 50, 50);
}

void mouseClicked() {
  if (value == 0) {
    value = 255;
  } else {
    value = 0;
  }
}
```

Description The **mouseClicked()** function is called once after a mouse button has been pressed and then released.

Syntax
```
mouseClicked()
mouseClicked(event)
```

Returns void

Related
mouseX
mouseY
mouseButton

mousePressed
mouseReleased()
mouseMoved()
mouseDragged()

Name

mouseDragged()

Examples

```
// Drag (click and hold) your mouse across the
// image to change the value of the rectangle

int value = 0;

void draw() {
  fill(value);
  rect(25, 25, 50, 50);
}

void mouseDragged()
{
  value = value + 5;
  if (value > 255) {
    value = 0;
  }
}
```

Description The **mouseDragged()** function is called once every time the mouse moves and a mouse button is pressed.

Syntax

```
mouseDragged()
mouseDragged(event)
```

Returns void

Related

mouseX
mouseY
mousePressed
mousePressed
mouseReleased()
mouseMoved()

Name

mouseMoved()

Examples

```
// Move the mouse across the image
// to change its value

int value = 0;

void draw() {
  fill(value);
  rect(25, 25, 50, 50);
}

void mouseMoved() {
  value = value + 5;
  if (value > 255) {
    value = 0;
  }
```

```
}
```

Description The **mouseMoved()** function is called every time the mouse moves and a mouse button is not pressed.

Syntax
```
mouseMoved()
mouseMoved(event)
```

Returns void

Related
mouseX
mouseY
mousePressed
mousePressed
mouseReleased()
mouseDragged()

Name

mousePressed()

Examples
```
// Click within the image to change
// the value of the rectangle

int value = 0;

void draw() {
  fill(value);
  rect(25, 25, 50, 50);
}

void mousePressed() {
  if (value == 0) {
    value = 255;
  } else {
    value = 0;
  }
}
```

Description The **mousePressed()** function is called once after every time a mouse button is pressed. The **mouseButton** variable (see the related reference entry) can be used to determine which button has been pressed.

Syntax
```
mousePressed()
mousePressed(event)
```

Returns void

Related
mouseX
mouseY
mousePressed
mouseButton
mouseReleased()
mouseMoved()
mouseDragged()

Name

mousePressed

Examples
```
// Click within the image to change
// the value of the rectangle
void draw() {
  if (mousePressed == true) {
```

```
    fill(0);
  } else {
    fill(255);
  }
  rect(25, 25, 50, 50);
}
```

Description Variable storing if a mouse button is pressed. The value of the system variable **mousePressed** is true if a mouse button is pressed and false if a button is not pressed.

Related
mouseX
mouseY
mouseReleased()
mouseMoved()
mouseDragged()

Name

mouseReleased()

Examples

```
// Click within the image to change
// the value of the rectangle

int value = 0;

void draw() {
  fill(value);
  rect(25, 25, 50, 50);
}

void mouseReleased() {
  if (value == 0) {
    value = 255;
  } else {
    value = 0;
  }
}
```

Description The **mouseReleased()** function is called every time a mouse button is released.

Syntax
```
mouseReleased()
mouseReleased(event)
```

Returns void

Related
mouseX
mouseY
mousePressed
mouseButton
mousePressed
mouseMoved()
mouseDragged()

Name

mouseWheel()

Examples

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

void draw() {}
```

```
void mouseWheel(MouseEvent event) {
  float e = event.getAmount();
  println(e);
}
```

Description The event.getAmount() method returns negative values if the mouse wheel if rotated up or away from the user and positive in the other direction. On OS X with "natural" scrolling enabled, the values are opposite.

Syntax `mouseWheel(event)`

Parameters **event** MouseEvent: the MouseEvent

Returns void

Name

mouseX

Examples

```
void draw()
{
  background(204);
  line(mouseX, 20, mouseX, 80);
}
```

Description The system variable **mouseX** always contains the current horizontal coordinate of the mouse.

Related
mouseY
mousePressed
mousePressed
mouseReleased()
mouseMoved()
mouseDragged()

Name

mouseY

Examples

```
void draw()
{
  background(204);
  line(20, mouseY, 80, mouseY);
}
```

Description The system variable **mouseY** always contains the current vertical coordinate of the mouse.

Related
mouseX
mousePressed
mousePressed
mouseReleased()
mouseMoved()
mouseDragged()

Name

pmouseX

Examples

```
// Move the mouse quickly to see the difference
// between the current and previous position
```

```
void draw() {
  background(204);
  line(mouseX, 20, pmouseX, 80);
  println(mouseX + " : " + pmouseX);
}
```

The system variable **pmouseX** always contains the horizontal position of the mouse in the frame previous to the current frame.

Description

You may find that **pmouseX** and **pmouseY** have different values inside **draw()** and inside events like **mousePressed()** and **mouseMoved()**. This is because they're used for different roles, so don't mix them. Inside **draw()**, **pmouseX** and **pmouseY** update only once per frame (once per trip through your **draw()**). But, inside mouse events, they update each time the event is called. If they weren't separated, then the mouse would be read only once per frame, making response choppy. If the mouse variables were always updated multiple times per frame, using **line(pmouseX, pmouseY, mouseX, mouseY)** inside **draw()** would have lots of gaps, because **pmouseX** may have changed several times in between the calls to **line()**. Use **pmouseX** and **pmouseY** inside **draw()** if you want values relative to the previous frame. Use **pmouseX** and **pmouseY** inside the mouse functions if you want continuous response.

Related

pmouseY
mouseX
mouseY

Name

pmouseY

Examples

```
// Move the mouse quickly to see the difference
// between the current and previous position
void draw() {
  background(204);
  line(20, mouseY, 80, pmouseY);
  println(mouseY + " : " + pmouseY);
}
```

Description

The system variable **pmouseY** always contains the vertical position of the mouse in the frame previous to the current frame. More detailed information about how **pmouseY** is updated inside of **draw()** and mouse events is explained in the reference for **pmouseX**.

Related

pmouseX
mouseX
mouseY

Name

key

Examples

```
// Click on the window to give it focus,
// and press the 'B' key.

void draw() {
  if (keyPressed) {
    if (key == 'b' || key == 'B') {
      fill(0);
    }
```

```
  } else {
    fill(255);
  }
  rect(25, 25, 50, 50);
}
```

Description

The system variable **key** always contains the value of the most recent key on the keyboard that was used (either pressed or released).

For non-ASCII keys, use the **keyCode** variable. The keys included in the ASCII specification (BACKSPACE, TAB, ENTER, RETURN, ESC, and DELETE) do not require checking to see if they key is coded, and you should simply use the **key** variable instead of **keyCode** If you're making cross-platform projects, note that the ENTER key is commonly used on PCs and Unix and the RETURN key is used instead on Macintosh. Check for both ENTER and RETURN to make sure your program will work for all platforms.

Related

keyCode
keyPressed
keyPressed
keyReleased()

Name

keyCode

Examples

```
color fillVal = color(126);

void draw() {
  fill(fillVal);
  rect(25, 25, 50, 50);
}

void keyPressed() {
  if (key == CODED) {
    if (keyCode == UP) {
      fillVal = 255;
    } else if (keyCode == DOWN) {
      fillVal = 0;
    }
  } else {
    fillVal = 126;
  }
}
```

The variable **keyCode** is used to detect special keys such as the UP, DOWN, LEFT, RIGHT arrow keys and ALT, CONTROL, SHIFT. When checking for these keys, it's first necessary to check and see if the key is coded. This is done with the conditional "if (key == CODED)" as shown in the example.

Description

The keys included in the ASCII specification (BACKSPACE, TAB, ENTER, RETURN, ESC, and DELETE) do not require checking to see if they key is coded, and you should simply use the **key** variable instead of **keyCode** If you're making cross-platform projects, note that the ENTER key is commonly used on PCs and Unix and the RETURN key is used instead on Macintosh. Check for both ENTER and RETURN to make sure your program will work for all platforms.

For users familiar with Java, the values for UP and DOWN are simply shorter versions of Java's KeyEvent.VK_UP and KeyEvent.VK_DOWN. Other keyCode values can be found in the Java KeyEvent reference.

Related
key
keyPressed
keyPressed
keyReleased()

Name

keyPressed()

Examples

```
// Click on the image to give it focus,
// and then press any key.

int value = 0;

void draw() {
  fill(value);
  rect(25, 25, 50, 50);
}

void keyPressed() {
  if (value == 0) {
    value = 255;
  } else {
    value = 0;
  }
}
```

Description

The **keyPressed()** function is called once every time a key is pressed. The key that was pressed is stored in the **key** variable.

For non-ASCII keys, use the **keyCode** variable. The keys included in the ASCII specification (BACKSPACE, TAB, ENTER, RETURN, ESC, and DELETE) do not require checking to see if they key is coded, and you should simply use the **key** variable instead of **keyCode** If you're making cross-platform projects, note that the ENTER key is commonly used on PCs and Unix and the RETURN key is used instead on Macintosh. Check for both ENTER and RETURN to make sure your program will work for all platforms.

Because of how operating systems handle key repeats, holding down a key may cause multiple calls to keyPressed() (and keyReleased() as well). The rate of repeat is set by the operating system and how each computer is configured.

Syntax

```
keyPressed()
keyPressed(event)
```

Returns void

Related
key
keyCode
keyPressed
keyReleased()

Name

keyPressed

```
// Click on the image to give it focus,
// and then press any key.

// Note: The rectangle in this example may
// flicker as the operating system may
// register a long key press as a repetition
// of key presses.
```

Examples

```
void draw() {
  if (keyPressed == true) {
    fill(0);
  } else {
    fill(255);
  }
  rect(25, 25, 50, 50);
}
```

Description The boolean system variable **keyPressed** is **true** if any key is pressed and **false** if no keys are pressed.

Related
key
keyCode
keyPressed
keyReleased()

Name

keyReleased()

```
// Click on the image to give it focus,
// and then press any key.

int value = 0;

void draw() {
  fill(value);
  rect(25, 25, 50, 50);
}

void keyReleased() {
  if (value == 0) {
    value = 255;
  } else {
    value = 0;
  }
}
```

Examples

Description The **keyReleased()** function is called once every time a key is released. The key that was released will be stored in the **key** variable. See **key** and **keyReleased** for more information.

Syntax
```
keyReleased()
keyReleased(event)
```

Returns void

Related
key
keyCode
keyPressed
keyPressed

Name

keyTyped()

Examples

```
// Run this program to learn how each of these functions
// relate to the others.

void draw() { } // Empty draw() needed to keep the program
running

void keyPressed() {
  println("pressed " + int(key) + " " + keyCode);
}

void keyTyped() {
  println("typed " + int(key) + " " + keyCode);
}

void keyReleased() {
  println("released " + int(key) + " " + keyCode);
}
```

Description The **keyTyped()** function is called once every time a key is pressed, but action keys such as Ctrl, Shift, and Alt are ignored. Because of how operating systems handle key repeats, holding down a key will cause multiple calls to **keyTyped()**, the rate is set by the operating system and how each computer is configured.

Syntax

```
keyTyped()
keyTyped(event)
```

Returns void

Related

keyPressed
key
keyCode
keyReleased()

Name

BufferedReader

Examples

```
BufferedReader reader;
String line;

void setup() {
  // Open the file from the createWriter() example
  reader = createReader("positions.txt");
}

void draw() {
  try {
    line = reader.readLine();
  } catch (IOException e) {
    e.printStackTrace();
    line = null;
  }
  if (line == null) {
    // Stop reading because of an error or file is empty
    noLoop();
  } else {
    String[] pieces = split(line, TAB);
    int x = int(pieces[0]);
```

```
    int y = int(pieces[1]);
    point(x, y);
  }
}
```

A **BufferedReader** object is used to read files line-by-line as individual **String** objects.

Description Starting with Processing release 0134, all files loaded and saved by the Processing API use UTF-8 encoding. In previous releases, the default encoding for your platform was used, which causes problems when files are moved to other platforms.

Parameters

Related createReader()
try
catch

Name

createInput()

Examples

```
// Load the local file 'data.txt' and initialize a new
InputStream
InputStream input = createInput("data.txt");

String content = "";

try {
  int data = input.read();
  while (data != -1) {
    content += data;
    data = input.read();
  }
}
catch (IOException e) {
  e.printStackTrace();
}
finally {
  try {
    input.close();
  }
  catch (IOException e) {
    e.printStackTrace();
  }
}

println(content);
```

Description This is a shorthand function for advanced programmers to initialize and open a Java InputStream. It's useful if you want to use the facilities provided by PApplet to easily open files from the data folder or from a URL, but you need an InputStream object so that you can use other parts of Java to take more control of how the stream is read.

The filename passed in can be:
- A URL, as in: **createInput("http://processing.org/")**
- The name of a file in the sketch's **data** folder
- The full path to a file to be opened locally (when running as an application)

If the requested item doesn't exist, null is returned. If not online, this will also check to see if the user is asking for a file whose name isn't properly capitalized. If capitalization is different, an error will be printed to the console. This helps prevent issues that appear when a sketch is exported to the web, where case sensitivity matters, as opposed to running from inside the Processing Development Environment on Windows or Mac OS, where case sensitivity is preserved but ignored.

If the file ends with **.gz**, the stream will automatically be gzip decompressed. If you don't want the automatic decompression, use the related function **createInputRaw()**.

In earlier releases, this function was called **openStream()**.

Syntax `createInput(filename)`

Parameters **filename** String: the name of the file to use as input

Returns InputStream

Related createOutput()

Name

createReader()

Examples

```
BufferedReader reader;
String line;

void setup() {
  // Open the file from the createWriter() example
  reader = createReader("positions.txt");
}

void draw() {
  try {
    line = reader.readLine();
  } catch (IOException e) {
    e.printStackTrace();
    line = null;
  }
  if (line == null) {
    // Stop reading because of an error or file is empty
    noLoop();
  } else {
    String[] pieces = split(line, TAB);
    int x = int(pieces[0]);
    int y = int(pieces[1]);
    point(x, y);
  }
}
```

Creates a **BufferedReader** object that can be used to read files line-by-line as individual **String** objects. This is the complement to the **createWriter()** function.

Description Starting with Processing release 0134, all files loaded and saved by the Processing API use UTF-8 encoding. In previous releases, the default encoding for your platform was used, which causes problems when files are moved to other platforms.

Syntax `createReader(filename)`
Parameters **filename** String: name of the file to be opened
Returns BufferedReader
Related createWriter()

Name

loadBytes()

Examples

```
// Open a file and read its binary data
byte b[] = loadBytes("something.dat");

// Print each value, from 0 to 255
for (int i = 0; i < b.length; i++) {
  // Every tenth number, start a new line
  if ((i % 10) == 0) {
    println();
  }
  // bytes are from -128 to 127, this converts to 0 to 255
  int a = b[i] & 0xff;
  print(a + " ");
}
// Print a blank line at the end
println();
```

Description Reads the contents of a file and places it in a byte array. If the name of the file is used as the parameter, as in the above example, the file must be loaded in the sketch's "data" directory/folder.

Alternatively, the file maybe be loaded from anywhere on the local computer using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows), or the filename parameter can be a URL for a file found on a network.

If the file is not available or an error occurs, **null** will be returned and an error message will be printed to the console. The error message does not halt the program, however the null value may cause a NullPointerException if your code does not check whether the value returned is null.

Syntax `loadBytes(filename)`
Parameters **filename** String: name of a file in the data folder or a URL.
Returns byte[]
Related loadStrings()
saveStrings()
saveBytes()

Name

loadJSONArray()

Examples

```
// The following short JSON file called "data.json" is parsed
// in the code below. It must be in the project's "data" folder.
//
// [
//   {
//     "id": 0,
//     "species": "Capra hircus",
```

```
//     "name": "Goat"
//   },
//   {
//     "id": 1,
//     "species": "Panthera pardus",
//     "name": "Leopard"
//   },
//   {
//     "id": 2,
//     "species": "Equus zebra",
//     "name": "Zebra"
//   }
// ]

JSONArray values;

void setup() {

  values = loadJSONArray("data.json");

  for (int i = 0; i < values.size(); i++) {

    JSONObject animal = values.getJSONObject(i);

    int id = animal.getInt("id");
    String species = animal.getString("species");
    String name = animal.getString("name");

    println(id + ", " + species + ", " + name);
  }
}

// Sketch prints:
// 0, Capra hircus, Goat
// 1, Panthera pardus, Leopard
// 2, Equus zebra, Zebra
```

Description Loads an array of JSON objects from the data folder or a URL, and returns a **JSONArray**. Per standard JSON syntax, the array must be enclosed in a pair of hard brackets **[]**, and each object within the array must be separated by a comma.

All files loaded and saved by the Processing API use UTF-8 encoding.

Syntax `loadJSONArray(filename)`

Parameters **filename** String: name of a file in the data folder or a URL

Returns JSONArray

Related JSONObject
JSONArray
loadJSONObject()
saveJSONObject()
saveJSONArray()

Name

loadJSONObject()

Examples

```
// The following short JSON file called "data.json" is parsed
// in the code below. It must be in the project's "data" folder.
//
// {
```

```
//   "id": 0,
//   "species": "Panthera leo",
//   "name": "Lion"
// }

JSONObject json;

void setup() {

  json = loadJSONObject("data.json");

  int id = json.getInt("id");
  String species = json.getString("species");
  String name = json.getString("name");

  println(id + ", " + species + ", " + name);
}

// Sketch prints:
// 0, Panthera leo, Lion
```

Loads a JSON from the data folder or a URL, and returns a **JSONObject**.

Description

All files loaded and saved by the Processing API use UTF-8 encoding.

Syntax `loadJSONObject(filename)`

Parameters **filename** String: name of a file in the data folder or a URL

Returns JSONObject

JSONObject
JSONArray

Related loadJSONArray()
saveJSONObject()
saveJSONArray()

Name

loadStrings()

```
String lines[] = loadStrings("list.txt");
println("there are " + lines.length + " lines");
for (int i = 0 ; i < lines.length; i++) {
  println(lines[i]);
}
```

Examples

```
String lines[] =
loadStrings("http://processing.org/about/index.html");
println("there are " + lines.length + " lines");
for (int i = 0 ; i < lines.length; i++) {
  println(lines[i]);
}
```

Reads the contents of a file and creates a String array of its individual lines. If the name of the file is used as the parameter, as in the above example, the file must be loaded in the sketch's "data" directory/folder.

Description Alternatively, the file maybe be loaded from anywhere on the local computer using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows), or the filename parameter can be a URL for a file found on a network.

If the file is not available or an error occurs, **null** will be returned and an error message will be printed to the console. The error message does not halt the program, however the null value may cause a NullPointerException if your code does not check whether the value returned is null.

Starting with Processing release 0134, all files loaded and saved by the Processing API use UTF-8 encoding. In previous releases, the default encoding for your platform was used, which causes problems when files are moved to other platforms.

Syntax

```
loadStrings(filename)
loadStrings(reader)
```

Parameters **filename** String: name of the file or url to load

Returns String[]

Related loadBytes()
saveStrings()
saveBytes()

Name

loadTable()

Examples

```
// The following short CSV file called "mammals.csv" is parsed
// in the code below. It must be in the project's "data" folder.
//
// id,species,name
// 0,Capra hircus,Goat
// 1,Panthera pardus,Leopard
// 2,Equus zebra,Zebra

Table table;

void setup() {

  table = loadTable("mammals.csv", "header");

  println(table.getRowCount() + " total rows in table");

  for (TableRow row : table.rows()) {

    int id = row.getInt("id");
    String species = row.getString("species");
    String name = row.getString("name");

    println(name + " (" + species + ") has an ID of " + id);
  }

}

// Sketch prints:
// 3 total rows in table
// Goat (Capra hircus) has an ID of 0
// Leopard (Panthera pardus) has an ID of 1
// Zebra (Equus zebra) has an ID of 2
```

Description Reads the contents of a file or URL and creates an Table object with its values. If a file is specified, it must be located in the sketch's "data" folder. The filename parameter can also be a URL to a file found online. By default, the file is assumed to be comma-separated (in CSV format). To use tab-separated data,

include "tsv" in the options parameter.

If the file contains a header row, include "header" in the options parameter. If the file does not have a header row, then simply omit the "header" option.

When specifying both a header and the file type, separate the options with commas, as in: **loadTable("data.csv", "header, tsv")**

All files loaded and saved by the Processing API use UTF-8 encoding.

Syntax

```
loadTable(filename)
loadTable(filename, options)
```

Parameters

filename String: name of a file in the data folder or a URL.

options String: may contain "header", "tsv", "csv", or "bin" separated by commas

Returns Table

Related

Table
saveTable()
loadBytes()
loadStrings()
loadXML()

Name

loadXML()

Examples

```
// The following short XML file called "mammals.xml" is parsed
// in the code below. It must be in the project's "data" folder.
//
// <?xml version="1.0"?>
// <mammals>
//   <animal id="0" species="Capra hircus">Goat</animal>
//   <animal id="1" species="Panthera pardus">Leopard</animal>
//   <animal id="2" species="Equus zebra">Zebra</animal>
// </mammals>

XML xml;

void setup() {
  xml = loadXML("mammals.xml");
  XML[] children = xml.getChildren("animal");

  for (int i = 0; i < children.length; i++) {
    int id = children[i].getInt("id");
    String coloring = children[i].getString("species");
    String name = children[i].getContent();
    println(id + ", " + coloring + ", " + name);
  }
}

// Sketch prints:
// 0, Capra hircus, Goat
// 1, Panthera pardus, Leopard
// 2, Equus zebra, Zebra
```

Description

Reads the contents of a file or URL and creates an XML object with its values. If a file is specified, it must be located in the sketch's "data" folder. The filename parameter can also be a URL to a file found online.

All files loaded and saved by the Processing API use UTF-8 encoding.

Syntax `loadXML(filename)`

Parameters **filename** String: name of a file in the data folder or a URL.

Returns XML

Related
XML
parseXML()
saveXML()
loadBytes()
loadStrings()
loadTable()

Name

open()

```
void setup() {
  size(200, 200);
}

void draw() {
  // draw() must be present for mousePressed() to work
}

void mousePressed() {
  println("Opening Process_4");
  open("/Applications/Process_4.app");
}
```

Examples

```
void setup() {
  size(200, 200);
}

void draw() {
  // draw() must be present for mousePressed() to work
}

void mousePressed() {
  String[] params = { "/usr/bin/jikes", "-help" };
  open(params);
}
```

Description

Attempts to open an application or file using your platform's launcher. The **filename** parameter is a String specifying the file name and location. The location parameter must be a full path name, or the name of an executable in the system's PATH. In most cases, using a full path is the best option, rather than relying on the system PATH. Be sure to make the file executable before attempting to open it (chmod +x).

The **argv** parameter is a String or String array which is passed to the command line. If you have multiple parameters, e.g. an application and a document, or a command with multiple switches, use the version that takes a String array, and place each individual item in a separate element.

If **argv** is a String (not an array), then it can only be a single file or application

with no parameters. It's not the same as executing that String using a shell. For instance, **open("jikes -help")** will not work properly.

This function behaves differently on each platform. On Windows, the parameters are sent to the Windows shell via "cmd /c". On Mac OS X, the "open" command is used (type "man open" in Terminal.app for documentation). On Linux, it first tries gnome-open, then kde-open, but if neither are available, it sends the command to the shell without any alterations.

For users familiar with Java, this is not quite the same as Runtime.exec(), because the launcher command is prepended. Instead, the **exec(String[])** function is a shortcut for Runtime.getRuntime.exec(String[]).

Syntax

```
open(filename)
open(argv)
```

Parameters **filename** String: name of the file
argv String[]: list of commands passed to the command line

Returns void or Process

Name

parseXML()

Examples

```
String data = "<mammals><animal>Goat</animal></mammals>";

void setup() {
  XML xml = parseXML(data);
  if (xml == null) {
    println("XML could not be parsed.");
  } else {
    XML firstChild = xml.getChild("animal");
    println(firstChild.getContent());
  }
}

// Sketch prints:
// Goat
```

Takes a String, parses its contents, and returns an XML object. If the String does not contain XML data or cannot be parsed, a null value is returned.

Description

parseXML() is most useful when pulling data dynamically, such as from third-party APIs. Normally, API results would be saved to a String, and then can be converted to a structured XML object using **parseXML()**. Be sure to check if null is returned before performing operations on the new XML object, in case the String content could not be parsed.

If your data already exists as an XML file in the data folder, it is simpler to use **loadXML()**.

Syntax

```
parseXML(xmlString)
parseXML(xmlString, options)
```

Returns XML

Related XML
loadXML()
saveXML()

Name

saveTable()

Examples

```
Table table;

void setup() {

  table = new Table();

  table.addColumn("id");
  table.addColumn("species");
  table.addColumn("name");

  TableRow newRow = table.addRow();
  newRow.setInt("id", table.getRowCount() - 1);
  newRow.setString("species", "Panthera leo");
  newRow.setString("name", "Lion");

  saveTable(table, "data/new.csv");
}

// Sketch saves the following to a file called "new.csv":
// id,species,name
// 0,Panthera leo,Lion
```

Description Writes the contents of a Table object to a file. By default, this file is saved to the sketch's folder. This folder is opened by selecting "Show Sketch Folder" from the "Sketch" menu.

Alternatively, the file can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

All files loaded and saved by the Processing API use UTF-8 encoding.

Syntax

```
saveTable(table, filename)
saveTable(table, filename, options)
```

Parameters

table Table: the Table object to save to a file
filename String: the filename to which the Table should be saved
options String: can be one of "tsv", "csv", "bin", or "html"

Returns boolean

Related Table
loadTable()

Name

selectFolder()

Examples

```
void setup() {
  selectFolder("Select a folder to process:", "folderSelected");
}

void folderSelected(File selection) {
  if (selection == null) {
    println("Window was closed or the user hit cancel.");
  } else {
    println("User selected " + selection.getAbsolutePath());
  }
```

```
}
```

Description Opens a platform-specific file chooser dialog to select a folder. After the selection is made, the selection will be passed to the 'callback' function. If the dialog is closed or canceled, null will be sent to the function, so that the program is not waiting for additional input. The callback is necessary because of how threading works.

Syntax

```
selectFolder(prompt, callback)
selectFolder(prompt, callback, file)
selectFolder(prompt, callback, file, callbackObject)
selectFolder(prompt, callbackMethod, defaultSelection,
callbackObject, parentFrame)
```

Parameters
prompt String: message to the user
callback String: name of the method to be called when the selection is made

Returns void

Name

selectInput()

Examples

```
void setup() {
  selectInput("Select a file to process:", "fileSelected");
}

void fileSelected(File selection) {
  if (selection == null) {
    println("Window was closed or the user hit cancel.");
  } else {
    println("User selected " + selection.getAbsolutePath());
  }
}
```

Description Opens a platform-specific file chooser dialog to select a file for input. After the selection is made, the selected File will be passed to the 'callback' function. If the dialog is closed or canceled, null will be sent to the function, so that the program is not waiting for additional input. The callback is necessary because of how threading works.

Syntax

```
selectInput(prompt, callback)
selectInput(prompt, callback, file)
selectInput(prompt, callback, file, callbackObject)
selectInput(prompt, callbackMethod, file, callbackObject, parent)
```

Parameters
prompt String: message to the user
callback String: name of the method to be called when the selection is made

Returns void

Name

day()

Examples

```
void setup() {
  PFont metaBold;
  metaBold = loadFont("fonts/Meta-Bold.vlw.gz");
  setFont(metaBold, 44);
  noLoop();
}

void draw() {
  int d = day();    // Values from 1 - 31
  int m = month();  // Values from 1 - 12
```

```
  int y = year();   // 2003, 2004, 2005, etc.
  String s = String.valueOf(d);
  text(s, 10, 28);
  s = String.valueOf(m);
  text(s, 10, 56);
  s = String.valueOf(y);
  text(s, 10, 84);
}
```

Description Processing communicates with the clock on your computer. The **day()** function returns the current day as a value from 1 - 31.

Syntax `day()`

Returns int

Related
millis()
second()
minute()
hour()
month()
year()

Name

hour()

Examples

```
void draw() {
  background(204);
  int s = second();  // Values from 0 - 59
  int m = minute();  // Values from 0 - 59
  int h = hour();    // Values from 0 - 23
  line(s, 0, s, 33);
  line(m, 33, m, 66);
  line(h, 66, h, 100);
}
```

Description Processing communicates with the clock on your computer. The **hour()** function returns the current hour as a value from 0 - 23.

Syntax `hour()`

Returns int

Related
millis()
second()
minute()
day()
month()
year()

Name

millis()

Examples

```
void draw() {
  int m = millis();
  noStroke();
  fill(m % 255);
  rect(25, 25, 50, 50);
}
```

Description Returns the number of milliseconds (thousandths of a second) since starting the program. This information is often used for timing events and animation

sequences.

Syntax `millis()`

Returns int

Related
- second()
- minute()
- hour()
- day()
- month()
- year()

Name

minute()

Examples

```
void draw() {
  background(204);
  int s = second();  // Values from 0 - 59
  int m = minute();  // Values from 0 - 59
  int h = hour();    // Values from 0 - 23
  line(s, 0, s, 33);
  line(m, 33, m, 66);
  line(h, 66, h, 100);
}
```

Description Processing communicates with the clock on your computer. The **minute()** function returns the current minute as a value from 0 - 59.

Syntax `minute()`

Returns int

Related
- millis()
- second()
- hour()
- day()
- month()
- year()

Name

month()

Examples

```
void setup() {
  PFont metaBold;
  metaBold = loadFont("fonts/Meta-Bold.vlw.gz");
  setFont(metaBold, 44);
  noLoop();
}

void draw() {
  int d = day();    // Values from 1 - 31
  int m = month();  // Values from 1 - 12
  int y = year();   // 2003, 2004, 2005, etc.
  String s = String.valueOf(d);
  text(s, 10, 28);
  s = String.valueOf(m);
  text(s, 10, 56);
  s = String.valueOf(y);
  text(s, 10, 84);
}
```

Description Processing communicates with the clock on your computer. The **month()** function returns the current month as a value from 1 - 12.

Syntax `month()`

Returns int

Related
millis()
second()
minute()
hour()
day()
year()

Name

second()

Examples

```
void draw() {
  background(204);
  int s = second();  // Values from 0 - 59
  int m = minute();  // Values from 0 - 59
  int h = hour();    // Values from 0 - 23
  line(s, 0, s, 33);
  line(m, 33, m, 66);
  line(h, 66, h, 100);
}
```

Description Processing communicates with the clock on your computer. The **second()** function returns the current second as a value from 0 - 59.

Syntax `second()`

Returns int

Related
millis()
minute()
hour()
day()
month()
year()

Name

year()

Examples

```
void setup() {
  PFont metaBold;
  metaBold = loadFont("fonts/Meta-Bold.vlw.gz");
  setFont(metaBold, 44);
  noLoop();
}

void draw() {
  int d = day();    // Values from 1 - 31
  int m = month();  // Values from 1 - 12
  int y = year();   // 2003, 2004, 2005, etc.
  String s = String.valueOf(d);
  text(s, 10, 28);
  s = String.valueOf(m);
  text(s, 10, 56);
  s = String.valueOf(y);
  text(s, 10, 84);
```

```
}
```

Description Processing communicates with the clock on your computer. The **year()** function returns the current year as an integer (2003, 2004, 2005, etc).

Syntax `year()`

Returns int

Related
millis()
second()
minute()
hour()
day()
month()

Name

print()

Examples

```
print("begin- ");
float f = 0.3;
int i = 1024;
print("f is " + f + " and i is " + 1024);
String s = " -end";
println(s);

// The above code prints:
// "begin- f is 0.3 and i is 1024 -end"
```

Writes to the console area of the Processing environment. This is often helpful for looking at the data a program is producing. The companion function **println()** works like **print()**, but creates a new line of text for each call to the function. Individual elements can be separated with quotes ("") and joined with the addition operator (+).

Description

Beginning with release 0125, to print the contents of an array, use println(). There's no sensible way to do a **print()** of an array, because there are too many possibilities for how to separate the data (spaces, commas, etc). If you want to print an array as a single line, use **join()**. With **join()**, you can choose any delimiter you like and **print()** the result.

Using **print()** on an object will output **null**, a memory location that may look like "@10be08," or the result of the **toString()** method from the object that's being printed. Advanced users who want more useful output when calling **print()** on their own classes can add a **toString()** method to the class that returns a String.

Syntax `print(what)`

Parameters **what** byte: boolean, byte, char, color, int, float, String, Object

Returns void

Related
println()
join()

Name

println()

Examples

```
println("begin");
float f = 0.3;
```

```
println("f is equal to " + f + " and i is equal to " + 1024);
String s = "end";
println(s);

// The above code prints the following lines:
// "begin"
// "f is equal to 0.3 and i is equal to 1024"
// "end"
```

```
float[] f = { 0.3, 0.4, 0.5 };
println(f);

// The above code prints:
// 0.3
// 0.4
// 0.5
```

Writes to the text area of the Processing environment's console. This is often helpful for looking at the data a program is producing. Each call to this function creates a new line of output. Individual elements can be separated with quotes ("") and joined with the string concatenation operator (+). See **print()** for more about what to expect in the output.

Description

Calling **println()** on an array (by itself) will write the contents of the array to the console. This is often helpful for looking at the data a program is producing. A new line is put between each element of the array. This function can only print one dimensional arrays. For arrays with higher dimensions, the result will be closer to that of **print()**.

Syntax

```
println()
println(what)
```

Parameters **what** byte: boolean, byte, char, color, int, float, String, Object

Returns void

Related print()

Name

save()

Examples

```
line(20, 20, 80, 80);
// Saves a TIFF file named "diagonal.tif"
save("diagonal.tif");
// Saves a TARGA file named "cross.tga"
line(80, 20, 20, 80);
save("cross.tga");
```

Description

Saves an image from the display window. Append a file extension to the name of the file, to indicate the file format to be used: either TIFF (.tif), TARGA (.tga), JPEG (.jpg), or PNG (.png). If no extension is included in the filename, the image will save in TIFF format and **.tif** will be added to the name. These files are saved to the sketch's folder, which may be opened by selecting "Show sketch folder" from the "Sketch" menu. Alternatively, the files can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

All images saved from the main drawing window will be opaque. To save images without a background, use **createGraphics()**.

Syntax

```
save(filename)
```

Parameters **filename** String: any sequence of letters and numbers

Returns void

Related saveFrame()
createGraphics()

Name

saveFrame()

```
int x = 0;
void draw()
{
  background(204);
  if (x < 100) {
    line(x, 0, x, 100);
    x = x + 1;
  } else {
    noLoop();
  }
  // Saves each frame as screen-0000.tif, screen-0001.tif, etc.
  saveFrame();
}
```

Examples

```
int x = 0;
void draw()
{
  background(204);
  if (x < 100) {
    line(x, 0, x, 100);
    x = x + 1;
  } else {
    noLoop();
  }
  // Saves each frame as line-000000.png, line-000001.png, etc.
  saveFrame("line-######.png");
}
```

Description

Saves a numbered sequence of images, one image each time the function is run. To save an image that is identical to the display window, run the function at the end of **draw()** or within mouse and key events such as **mousePressed()** and **keyPressed()**. Use the Movie Maker program in the Tools menu to combine these images to a movie.

If **saveFrame()** is used without parameters, it will save files as screen-0000.tif, screen-0001.tif, and so on. You can specify the name of the sequence with the **filename** parameter, including hash marks (####), which will be replaced by the saved frame count value. (The number of hash marks is used to determine how many digits to include in the file names.) Append a file extension, to indicate the file format to be used: either TIFF (.tif), TARGA (.tga), JPEG (.jpg), or PNG (.png). Image files are saved to the sketch's folder, which may be opened by selecting "Show Sketch Folder" from the "Sketch" menu.

Alternatively, the files can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

All images saved from the main drawing window will be opaque. To save images

without a background, use **createGraphics()**.

Syntax

```
saveFrame()
saveFrame(filename)
```

Parameters **filename** String: any sequence of letters or numbers that ends with either ".tif", ".tga", ".jpg", or ".png"

Returns void

Related save()
createGraphics()

Name

beginRaw()

Examples

```
import processing.pdf.*;

void setup() {
  size(400, 400);
  beginRaw(PDF, "raw.pdf");
}

void draw() {
  line(pmouseX, pmouseY, mouseX, mouseY);
}

void keyPressed() {
  if (key == ' ') {
    endRaw();
    exit();
  }
}
```

Description

To create vectors from 3D data, use the **beginRaw()** and **endRaw()** commands. These commands will grab the shape data just before it is rendered to the screen. At this stage, your entire scene is nothing but a long list of individual lines and triangles. This means that a shape created with **sphere()** function will be made up of hundreds of triangles, rather than a single object. Or that a multi-segment line shape (such as a curve) will be rendered as individual segments.

When using **beginRaw()** and **endRaw()**, it's possible to write to either a 2D or 3D renderer. For instance, **beginRaw()** with the PDF library will write the geometry as flattened triangles and lines, even if recording from the **P3D** renderer.

If you want a background to show up in your files, use **rect(0, 0, width, height)** after setting the **fill()** to the background color. Otherwise the background will not be rendered to the file because the background is not shape.

Using **hint(ENABLE_DEPTH_SORT)** can improve the appearance of 3D geometry drawn to 2D file formats. See the **hint()** reference for more details.

See examples in the reference for the **PDF** and **DXF** libraries for more information.

Syntax

```
beginRaw(renderer, filename)
```

Parameters **renderer** String: for example, PDF or DXF
filename String: filename for output

Returns PGraphics or void
Related endRaw()

Name

beginRecord()

```
import processing.pdf.*;

void setup() {
  size(400, 400);
  beginRecord(PDF, "everything.pdf");
}
```

Examples

```
void draw() {
  ellipse(mouseX, mouseY, 10, 10);
}

void mousePressed() {
  endRecord();
  exit();
}
```

Description Opens a new file and all subsequent drawing functions are echoed to this file as well as the display window. The **beginRecord()** function requires two parameters, the first is the renderer and the second is the file name. This function is always used with **endRecord()** to stop the recording process and close the file.

Note that beginRecord() will only pick up any settings that happen after it has been called. For instance, if you call textFont() before beginRecord(), then that font will not be set for the file that you're recording to.

Syntax

```
beginRecord(renderer, filename)
```

Parameters **renderer** String: for example, PDF
filename String: filename for output

Returns PGraphics or void
Related endRecord()

Name

createOutput()

Examples

Description Similar to **createInput()**, this creates a Java **OutputStream** for a given filename or path. The file will be created in the sketch folder, or in the same folder as an exported application.

If the path does not exist, intermediate folders will be created. If an exception occurs, it will be printed to the console, and **null** will be returned.

This function is a convenience over the Java approach that requires you to 1) create a FileOutputStream object, 2) determine the exact file location, and 3) handle exceptions. Exceptions are handled internally by the function, which is more appropriate for "sketch" projects.

If the output filename ends with **.gz**, the output will be automatically GZIP

compressed as it is written.

Syntax `createOutput(filename)`

Parameters **filename** String: name of the file to open

Returns OutputStream

Related createInput()

Name

createWriter()

Examples

```
PrintWriter output;

void setup() {
  // Create a new file in the sketch directory
  output = createWriter("positions.txt");
}

void draw() {
  point(mouseX, mouseY);
  output.println(mouseX + "t" + mouseY); // Write the coordinate
to the file
}

void keyPressed() {
  output.flush(); // Writes the remaining data to the file
  output.close(); // Finishes the file
  exit(); // Stops the program
}
```

Description Creates a new file in the sketch folder, and a **PrintWriter** object to write to it. For the file to be made correctly, it should be flushed and must be closed with its **flush()** and **close()** methods (see above example).

Starting with Processing release 0134, all files loaded and saved by the Processing API use UTF-8 encoding. In previous releases, the default encoding for your platform was used, which causes problems when files are moved to other platforms.

Syntax `createWriter(filename)`

Parameters **filename** String: name of the file to be created

Returns PrintWriter

Related createReader()

Name

endRaw()

Examples

```
import processing.pdf.*;

void setup() {
  size(400, 400);
  beginRaw(PDF, "raw.pdf");
}

void draw() {
  line(pmouseX, pmouseY, mouseX, mouseY);
}
```

```
void keyPressed() {
  if (key == ' ') {
    endRaw();
    exit();
  }
}
```

Description Complement to **beginRaw()**; they must always be used together. See the **beginRaw()** reference for details.

Syntax `endRaw()`

Returns void

Related beginRaw()

Name

endRecord()

Examples

```
import processing.pdf.*;

void setup() {
  size(400, 400);
  beginRecord(PDF, "everything.pdf");
}

void draw() {
  ellipse(mouseX, mouseY, 10, 10);
}

void mousePressed() {
  endRecord();
  exit();
}
```

Description Stops the recording process started by **beginRecord()** and closes the file.

Syntax `endRecord()`

Returns void

Related beginRecord()

Name

PrintWriter

Examples

```
PrintWriter output;

void setup() {
  // Create a new file in the sketch directory
  output = createWriter("positions.txt");
}

void draw() {
  point(mouseX, mouseY);
  output.println(mouseX);  // Write the coordinate to the file
}

void keyPressed() {
  output.flush();  // Writes the remaining data to the file
  output.close();  // Finishes the file
  exit();  // Stops the program
}
```

Description Allows characters to print to a text-output stream. A new PrintWriter object is

created with the **createWriter()** function. For the file to be made correctly, it should be flushed and must be closed with its **flush()** and **close()** methods (see above example).

Methods
- **print()** Adds data to the stream
- **println()** Adds data to the stream and starts a new line
- **flush()** Flushes the stream
- **close()** Closes the stream

Constructor

Related createWriter()

Name

saveBytes()

Examples

```
byte[] nums = { 0, 34, 5, 127, 52};

// Writes the bytes to a file
saveBytes("numbers.dat", nums);
```

Description As the opposite of **loadBytes()**, this function will write an entire array of bytes to a file. The data is saved in binary format. This file is saved to the sketch's folder, which is opened by selecting "Show Sketch Folder" from the "Sketch" menu. Alternatively, the files can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

Syntax

```
saveBytes(filename, data)
```

Parameters
- **filename** String: name of the file to write to
- **data** byte[]: array of bytes to be written

Returns void

Related loadStrings()
loadBytes()
saveStrings()

Name

saveJSONArray()

Examples

```
String[] species = { "Capra hircus", "Panthera pardus", "Equus
zebra" };
String[] names = { "Goat", "Leopard", "Zebra" };

JSONArray values;

void setup() {

  values = new JSONArray();

  for (int i = 0; i < species.length; i++) {

    JSONObject animal = new JSONObject();

    animal.setInt("id", i);
    animal.setString("species", species[i]);
    animal.setString("name", names[i]);

    values.setJSONObject(i, animal);
```

```
  }

  saveJSONArray(values, "data/new.json");
}

// Sketch saves the following to a file called "new.json":
// [
//   {
//     "id": 0,
//     "species": "Capra hircus",
//     "name": "Goat"
//   },
//   {
//     "id": 1,
//     "species": "Panthera pardus",
//     "name": "Leopard"
//   },
//   {
//     "id": 2,
//     "species": "Equus zebra",
//     "name": "Zebra"
//   }
// ]
```

Writes the contents of a **JSONArray** object to a file. By default, this file is saved to the sketch's folder. This folder is opened by selecting "Show Sketch Folder" from the "Sketch" menu.

Description Alternatively, the file can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

All files loaded and saved by the Processing API use UTF-8 encoding.

Syntax

```
saveJSONArray(json, filename)
saveJSONArray(json, filename, options)
```

Returns boolean

Related JSONObject
JSONArray
loadJSONObject()
loadJSONArray()
saveJSONObject()

Name

saveJSONObject()

Examples

```
JSONObject json;

void setup() {

  json = new JSONObject();

  json.setInt("id", 0);
  json.setString("species", "Panthera leo");
  json.setString("name", "Lion");

  saveJSONObject(json, "data/new.json");
}
```

```
// Sketch saves the following to a file called "new.json":
// {
//   "id": 0,
//   "species": "Panthera leo",
//   "name": "Lion"
// }
```

Writes the contents of a **JSONObject** object to a file. By default, this file is saved to the sketch's folder. This folder is opened by selecting "Show Sketch Folder" from the "Sketch" menu.

Description Alternatively, the file can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

All files loaded and saved by the Processing API use UTF-8 encoding.

Syntax
```
saveJSONObject(json, filename)
saveJSONObject(json, filename, options)
```

Returns boolean

Related JSONObject
JSONArray
loadJSONObject()
loadJSONArray()
saveJSONArray()

Name

saveStream()

Examples

Save the contents of a stream to a file in the sketch folder. This is basically **saveBytes(blah, loadBytes())**, but done more efficiently (and with less confusing syntax).

Description

The **target** parameter can be either a String specifying a file name, or, for greater control over the file location, a **File** object. (Note that, unlike some other functions, this will not automatically compress or uncompress gzip files.)

Syntax
```
saveStream(target, source)
```

Parameters **target** File, or String: name of the file to write to
source String: location to read from (a filename, path, or URL)

Returns boolean or void

Related createOutput()

Name

saveStrings()

Examples

```
String words = "apple bear cat dog";
String[] list = split(words, ' ');
// Writes the strings to a file, each on a separate line
saveStrings("nouns.txt", list);
```

Description Writes an array of Strings to a file, one line per String. By default, this file is saved to the sketch's folder. This folder is opened by selecting "Show Sketch Folder" from the "Sketch" menu.

Alternatively, the file can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

Starting with Processing 1.0, all files loaded and saved by the Processing API use UTF-8 encoding. In earlier releases, the default encoding for your platform was used, which causes problems when files are moved to other platforms.

Syntax `saveStrings(filename, data)`

Parameters
- **filename** String: filename for output
- **data** String[]: string array to be written

Returns void

Related loadStrings()
loadBytes()
saveBytes()

Name

saveXML()

```
// The following short XML file called "mammals.xml" is parsed
// in the code below. It must be in the project's "data" folder.
//
// <?xml version="1.0"?>
// <mammals>
//   <animal id="0" species="Capra hircus">Goat</animal>
//   <animal id="1" species="Panthera pardus">Leopard</animal>
//   <animal id="2" species="Equus zebra">Zebra</animal>
// </mammals>

XML xml;
```

Examples

```
void setup() {
  xml = loadXML("mammals.xml");
  XML firstChild = xml.getChild("animal");
  xml.removeChild(firstChild);
  saveXML(xml, "subset.xml");
}

// Sketch saves the following to a file called "subset.xml":
// <?xml version="1.0"?>
// <mammals>
//   <animal id="1" species="Panthera pardus">Leopard</animal>
//   <animal id="2" species="Equus zebra">Zebra</animal>
// </mammals>
```

Writes the contents of an XML object to a file. By default, this file is saved to the sketch's folder. This folder is opened by selecting "Show Sketch Folder" from the "Sketch" menu.

Description Alternatively, the file can be saved to any location on the computer by using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows).

All files loaded and saved by the Processing API use UTF-8 encoding.

Syntax `saveXML(xml, filename)`

```
saveXML(xml, filename, options)
```

Parameters
- **xml** XML: the XML object to save to disk
- **filename** String: name of the file to write to

Returns boolean

Related XML, loadXML(), parseXML()

Name

selectOutput()

Examples

```
void setup() {
  selectOutput("Select a file to write to:", "fileSelected");
}

void fileSelected(File selection) {
  if (selection == null) {
    println("Window was closed or the user hit cancel.");
  } else {
    println("User selected " + selection.getAbsolutePath());
  }
}
```

Description Opens a platform-specific file chooser dialog to select a file for output. After the selection is made, the selected File will be passed to the 'callback' function. If the dialog is closed or canceled, null will be sent to the function, so that the program is not waiting for additional input. The callback is necessary because of how threading works.

Syntax

```
selectOutput(prompt, callback)
selectOutput(prompt, callback, file)
selectOutput(prompt, callback, file, callbackObject)
selectOutput(prompt, callbackMethod, file, callbackObject,
parent)
```

Parameters
- **prompt** String: message to the user
- **callback** String: name of the method to be called when the selection is made

Returns void

Name

applyMatrix()

Examples

```
size(100, 100, P3D);
noFill();
translate(50, 50, 0);
rotateY(PI/6);
stroke(153);
box(35);
// Set rotation angles
float ct = cos(PI/9.0);
float st = sin(PI/9.0);
// Matrix for rotation around the Y axis
applyMatrix(  ct, 0.0,  st,  0.0,
              0.0, 1.0, 0.0,  0.0,
              -st, 0.0,  ct,  0.0,
              0.0, 0.0, 0.0,  1.0);
```

```
stroke(255);
box(50);
```

Description Multiplies the current matrix by the one specified through the parameters. This is very slow because it will try to calculate the inverse of the transform, so avoid it whenever possible. The equivalent function in OpenGL is glMultMatrix().

Syntax
```
applyMatrix(source)
applyMatrix(n00, n01, n02, n10, n11, n12)
applyMatrix(n00, n01, n02, n03, n10, n11, n12, n13, n20, n21,
n22, n23, n30, n31, n32, n33)
```

Parameters
n00 float: numbers which define the 4x4 matrix to be multiplied
n01 float: numbers which define the 4x4 matrix to be multiplied
n02 float: numbers which define the 4x4 matrix to be multiplied
n10 float: numbers which define the 4x4 matrix to be multiplied
n11 float: numbers which define the 4x4 matrix to be multiplied
n12 float: numbers which define the 4x4 matrix to be multiplied
n03 float: numbers which define the 4x4 matrix to be multiplied
n13 float: numbers which define the 4x4 matrix to be multiplied
n20 float: numbers which define the 4x4 matrix to be multiplied
n21 float: numbers which define the 4x4 matrix to be multiplied
n22 float: numbers which define the 4x4 matrix to be multiplied
n23 float: numbers which define the 4x4 matrix to be multiplied
n30 float: numbers which define the 4x4 matrix to be multiplied
n31 float: numbers which define the 4x4 matrix to be multiplied
n32 float: numbers which define the 4x4 matrix to be multiplied
n33 float: numbers which define the 4x4 matrix to be multiplied

Returns void

Related
pushMatrix()
popMatrix()
resetMatrix()
printMatrix()

Name

popMatrix()

Examples
```
fill(255);
rect(0, 0, 50, 50);  // White rectangle

pushMatrix();
translate(30, 20);
fill(0);
rect(0, 0, 50, 50);  // Black rectangle
popMatrix();

fill(100);
rect(15, 10, 50, 50);  // Gray rectangle
```

Description Pops the current transformation matrix off the matrix stack. Understanding pushing and popping requires understanding the concept of a matrix stack. The **pushMatrix()** function saves the current coordinate system to the stack and **popMatrix()** restores the prior coordinate system. **pushMatrix()** and **popMatrix()** are used in conjuction with the other transformation functions and may be embedded to control the scope of the transformations.

Syntax `popMatrix()`
Returns void
Related pushMatrix()

Name

printMatrix()

Examples

```
size(100, 100, P3D);
printMatrix();
// Prints:
// 01.0000  00.0000  00.0000 -50.0000
// 00.0000  01.0000  00.0000 -50.0000
// 00.0000  00.0000  01.0000 -86.6025
// 00.0000  00.0000  00.0000  01.0000
```

```
resetMatrix();
printMatrix();
// Prints:
// 1.0000  0.0000  0.0000  0.0000
// 0.0000  1.0000  0.0000  0.0000
// 0.0000  0.0000  1.0000  0.0000
// 0.0000  0.0000  0.0000  1.0000
```

Description Prints the current matrix to the Console (the text window at the bottom of Processing).
Syntax `printMatrix()`
Returns void
Related pushMatrix()
popMatrix()
resetMatrix()
applyMatrix()

Name

pushMatrix()

Examples

```
fill(255);
rect(0, 0, 50, 50);  // White rectangle

pushMatrix();
translate(30, 20);
fill(0);
rect(0, 0, 50, 50);  // Black rectangle
popMatrix();

fill(100);
rect(15, 10, 50, 50);  // Gray rectangle
```

Description Pushes the current transformation matrix onto the matrix stack. Understanding **pushMatrix()** and **popMatrix()** requires understanding the concept of a matrix stack. The **pushMatrix()** function saves the current coordinate system to the stack and **popMatrix()** restores the prior coordinate system. **pushMatrix()** and **popMatrix()** are used in conjuction with the other transformation functions and may be embedded to control the scope of the transformations.
Syntax `pushMatrix()`

Returns void

Related
popMatrix()
translate()
rotate()
rotateX()
rotateY()
rotateZ()

Name

resetMatrix()

Examples

```
size(100, 100, P3D);
noFill();
box(80);
printMatrix();
// Prints:
// 01.0000  00.0000  00.0000 -50.0000
// 00.0000  01.0000  00.0000 -50.0000
// 00.0000  00.0000  01.0000 -86.6025
// 00.0000  00.0000  00.0000  01.0000

resetMatrix();
box(80);
printMatrix();
// Prints:
// 1.0000  0.0000  0.0000  0.0000
// 0.0000  1.0000  0.0000  0.0000
// 0.0000  0.0000  1.0000  0.0000
// 0.0000  0.0000  0.0000  1.0000
```

Description Replaces the current matrix with the identity matrix. The equivalent function in OpenGL is **glLoadIdentity()**.

Syntax `resetMatrix()`

Returns void

Related
pushMatrix()
popMatrix()
applyMatrix()
printMatrix()

Name

rotate()

Examples

```
translate(width/2, height/2);
rotate(PI/3.0);
rect(-26, -26, 52, 52);
```

Description Rotates a shape the amount specified by the **angle** parameter. Angles should be specified in radians (values from 0 to TWO_PI) or converted to radians with the **radians()** function.

Objects are always rotated around their relative position to the origin and positive

numbers rotate objects in a clockwise direction. Transformations apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling **rotate(HALF_PI)** and then **rotate(HALF_PI)** is the same as **rotate(PI)**. All tranformations are reset when **draw()** begins again.

Technically, **rotate()** multiplies the current transformation matrix by a rotation matrix. This function can be further controlled by the **pushMatrix()** and **popMatrix()**.

Syntax
```
rotate(angle)
rotate(angle, x, y, z)
```

Parameters **angle** float: angle of rotation specified in radians

Returns void

Related
popMatrix()
pushMatrix()
rotateX()
rotateY()
rotateZ()
scale()
radians()

Name

rotateX()

Examples

```
size(100, 100, P3D);
translate(width/2, height/2);
rotateX(PI/3.0);
rect(-26, -26, 52, 52);
```

```
size(100, 100, P3D);
translate(width/2, height/2);
rotateX(radians(60));
rect(-26, -26, 52, 52);
```

Description

Rotates a shape around the x-axis the amount specified by the **angle** parameter. Angles should be specified in radians (values from 0 to PI*2) or converted to radians with the **radians()** function. Objects are always rotated around their relative position to the origin and positive numbers rotate objects in a counterclockwise direction. Transformations apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling **rotateX(PI/2)** and then **rotateX(PI/2)** is the same as **rotateX(PI)**. If **rotateX()** is called within the **draw()**, the transformation is reset when the loop begins again. This function requires using P3D as a third parameter to **size()** as shown in the example above.

Syntax
```
rotateX(angle)
```

Parameters **angle** float: angle of rotation specified in radians

Returns void

Related
popMatrix()
pushMatrix()
rotate()
rotateY()

rotateZ()
scale()
translate()

Name

rotateY()

Examples

```
size(100, 100, P3D);
translate(width/2, height/2);
rotateY(PI/3.0);
rect(-26, -26, 52, 52);
```

```
size(100, 100, P3D);
translate(width/2, height/2);
rotateY(radians(60));
rect(-26, -26, 52, 52);
```

Description Rotates a shape around the y-axis the amount specified by the **angle** parameter. Angles should be specified in radians (values from 0 to PI*2) or converted to radians with the **radians()** function. Objects are always rotated around their relative position to the origin and positive numbers rotate objects in a counterclockwise direction. Transformations apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling **rotateY(PI/2)** and then **rotateY(PI/2)** is the same as **rotateY(PI)**. If **rotateY()** is called within the **draw()**, the transformation is reset when the loop begins again. This function requires using P3D as a third parameter to **size()** as shown in the examples above.

Syntax `rotateY(angle)`

Parameters **angle** float: angle of rotation specified in radians

Returns void

Related
popMatrix()
pushMatrix()
rotate()
rotateX()
rotateZ()
scale()
translate()

Name

rotateZ()

Examples

```
size(100, 100, P3D);
translate(width/2, height/2);
rotateZ(PI/3.0);
rect(-26, -26, 52, 52);
```

```
size(100, 100, P3D);
translate(width/2, height/2);
```

```
rotateZ(radians(60));
rect(-26, -26, 52, 52);
```

Description Rotates a shape around the z-axis the amount specified by the **angle** parameter. Angles should be specified in radians (values from 0 to PI*2) or converted to radians with the **radians()** function. Objects are always rotated around their relative position to the origin and positive numbers rotate objects in a counterclockwise direction. Transformations apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling **rotateZ(PI/2)** and then **rotateZ(PI/2)** is the same as **rotateZ(PI)**. If **rotateZ()** is called within the **draw()**, the transformation is reset when the loop begins again. This function requires using P3D as a third parameter to **size()** as shown in the examples above.

Syntax `rotateZ(angle)`

Parameters **angle** float: angle of rotation specified in radians

Returns void

Related
popMatrix()
pushMatrix()
rotate()
rotateX()
rotateY()
scale()
translate()

Name

scale()

```
rect(30, 20, 50, 50);
scale(0.5);
rect(30, 20, 50, 50);
```

```
rect(30, 20, 50, 50);
scale(0.5, 1.3);
rect(30, 20, 50, 50);
```

Examples

```
// Scaling in 3D requires P3D
// as a parameter to size()
size(100, 100, P3D);
noFill();
translate(width/2+12, height/2);
box(20, 20, 20);
scale(2.5, 2.5, 2.5);
box(20, 20, 20);
```

Description Increases or decreases the size of a shape by expanding and contracting vertices. Objects always scale from their relative origin to the coordinate system. Scale values are specified as decimal percentages. For example, the function call **scale(2.0)** increases the dimension of a shape by 200%.

Transformations apply to everything that happens after and subsequent calls to the function multiply the effect. For example, calling **scale(2.0)** and then **scale(1.5)** is the same as **scale(3.0)**. If **scale()** is called within **draw()**, the

transformation is reset when the loop begins again. Using this fuction with the **z** parameter requires using P3D as a parameter for **size()**, as shown in the third example above. This function can be further controlled with **pushMatrix()** and **popMatrix()**.

Syntax

```
scale(s)
scale(x, y)
scale(x, y, z)
```

Parameters

s float: percentage to scale the object
x float: percentage to scale the object in the x-axis
y float: percentage to scale the object in the y-axis
z float: percentage to scale the object in the z-axis

Returns void

Related

pushMatrix()
popMatrix()
translate()
rotate()
rotateX()
rotateY()
rotateZ()

Name

shearX()

Examples

```
size(100, 100);
translate(width/4, height/4);
shearX(PI/4.0);
rect(0, 0, 30, 30);
```

Description

Shears a shape around the x-axis the amount specified by the **angle** parameter. Angles should be specified in radians (values from 0 to PI*2) or converted to radians with the **radians()** function. Objects are always sheared around their relative position to the origin and positive numbers shear objects in a clockwise direction. Transformations apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling **shearX(PI/2)** and then **shearX(PI/2)** is the same as **shearX(PI)**. If **shearX()** is called within the **draw()**, the transformation is reset when the loop begins again.

Technically, **shearX()** multiplies the current transformation matrix by a rotation matrix. This function can be further controlled by the **pushMatrix()** and **popMatrix()** functions.

Syntax

```
shearX(angle)
```

Parameters

angle float: angle of shear specified in radians

Returns void

Related

popMatrix()
pushMatrix()
shearY()
scale()
translate()
radians()

Name

shearY()

Examples

```
size(100, 100);
translate(width/4, height/4);
shearY(PI/4.0);
rect(0, 0, 30, 30);
```

Description

Shears a shape around the y-axis the amount specified by the **angle** parameter. Angles should be specified in radians (values from 0 to PI*2) or converted to radians with the **radians()** function. Objects are always sheared around their relative position to the origin and positive numbers shear objects in a clockwise direction. Transformations apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling **shearY(PI/2)** and then **shearY(PI/2)** is the same as **shearY(PI)**. If **shearY()** is called within the **draw()**, the transformation is reset when the loop begins again.

Technically, **shearY()** multiplies the current transformation matrix by a rotation matrix. This function can be further controlled by the **pushMatrix()** and **popMatrix()** functions.

Syntax

```
shearY(angle)
```

Parameters **angle** float: angle of shear specified in radians

Returns void

Related

popMatrix()
pushMatrix()
shearX()
scale()
translate()
radians()

Name

translate()

Examples

```
translate(30, 20);
rect(0, 0, 55, 55);
```

```
// Translating in 3D requires P3D
// as the parameter to size()
size(100, 100, P3D);
// Translate 30 across, 20 down, and
// 50 back, or "away" from the screen.
translate(30, 20, -50);
rect(0, 0, 55, 55);
```

```
rect(0, 0, 55, 55);  // Draw rect at original 0,0
translate(30, 20);
rect(0, 0, 55, 55);  // Draw rect at new 0,0
translate(14, 14);
```

```
rect(0, 0, 55, 55);  // Draw rect at new 0,0
```

Specifies an amount to displace objects within the display window. The **x** parameter specifies left/right translation, the **y** parameter specifies up/down translation, and the **z** parameter specifies translations toward/away from the screen. Using this function with the **z** parameter requires using P3D as a parameter in combination with size as shown in the above example.

Description Transformations are cumulative and apply to everything that happens after and subsequent calls to the function accumulates the effect. For example, calling **translate(50, 0)** and then **translate(20, 0)** is the same as **translate(70, 0)**. If **translate()** is called within **draw()**, the transformation is reset when the loop begins again. This function can be further controlled by using **pushMatrix()** and **popMatrix()**.

Syntax
```
translate(x, y)
translate(x, y, z)
```

Parameters
x float: left/right translation
y float: up/down translation
z float: forward/backward translation

Returns void

Related
popMatrix()
pushMatrix()
rotate()
rotateX()
rotateY()
rotateZ()
scale()

Name

ambientLight()

Examples

```
size(100, 100, P3D);
background(0);
noStroke();
// The spheres are white by default so
// the ambient light changes their color
ambientLight(51, 102, 126);
translate(20, 50, 0);
sphere(30);
translate(60, 0, 0);
sphere(30);
```

```
size(100, 100, P3D);
background(0);
noStroke();
directionalLight(126, 126, 126, 0, 0, -1);
ambientLight(102, 102, 102);
translate(32, 50, 0);
rotateY(PI/5);
box(40);
translate(60, 0, 0);
sphere(30);
```

Description Adds an ambient light. Ambient light doesn't come from a specific direction, the

rays have light have bounced around so much that objects are evenly lit from all sides. Ambient lights are almost always used in combination with other types of lights. Lights need to be included in the **draw()** to remain persistent in a looping program. Placing them in the **setup()** of a looping program will cause them to only have an effect the first time through the loop. The **v1**, **v2**, and **v3** parameters are interpreted as either RGB or HSB values, depending on the current color mode.

Syntax

```
ambientLight(v1, v2, v3)
ambientLight(v1, v2, v3, x, y, z)
```

Parameters

v1 float: red or hue value (depending on current color mode)
v2 float: green or saturation value (depending on current color mode)
v3 float: blue or brightness value (depending on current color mode)
x float: x-coordinate of the light
y float: y-coordinate of the light
z float: z-coordinate of the light

Returns void

Related lights()
directionalLight()
pointLight()
spotLight()

Name

directionalLight()

Examples

```
size(100, 100, P3D);
background(0);
noStroke();
directionalLight(51, 102, 126, -1, 0, 0);
translate(20, 50, 0);
sphere(30);
```

```
size(100, 100, P3D);
background(0);
noStroke();
directionalLight(51, 102, 126, 0, -1, 0);
translate(80, 50, 0);
sphere(30);
```

Description

Adds a directional light. Directional light comes from one direction: it is stronger when hitting a surface squarely, and weaker if it hits at a gentle angle. After hitting a surface, directional light scatters in all directions. Lights need to be included in the **draw()** to remain persistent in a looping program. Placing them in the **setup()** of a looping program will cause them to only have an effect the first time through the loop. The **v1**, **v2**, and **v3** parameters are interpreted as either RGB or HSB values, depending on the current color mode. The **nx**, **ny**, and **nz** parameters specify the direction the light is facing. For example, setting **ny** to -1 will cause the geometry to be lit from below (since the light would be facing directly upward).

Syntax

```
directionalLight(v1, v2, v3, nx, ny, nz)
```

Parameters

v1 float: red or hue value (depending on current color mode)

v2 float: green or saturation value (depending on current color mode)
v3 float: blue or brightness value (depending on current color mode)
nx float: direction along the x-axis
ny float: direction along the y-axis
nz float: direction along the z-axis

Returns void

Related lights()
ambientLight()
pointLight()
spotLight()

Name

lightFalloff()

Examples

```
size(100, 100, P3D);
noStroke();
background(0);
lightFalloff(1.0, 0.001, 0.0);
pointLight(150, 250, 150, 50, 50, 50);
beginShape();
vertex(0, 0, 0);
vertex(100, 0, -100);
vertex(100, 100, -100);
vertex(0, 100, 0);
endShape(CLOSE);
```

Description Sets the falloff rates for point lights, spot lights, and ambient lights. Like **fill()**, it affects only the elements which are created after it in the code. The default value is **lightFalloff(1.0, 0.0, 0.0)**, and the parameters are used to calculate the falloff with the following equation:

d = distance from light position to vertex position
falloff = 1 / (CONSTANT + d * LINEAR + (d*d) * QUADRATIC)

Thinking about an ambient light with a falloff can be tricky. If you want a region of your scene to be lit ambiently with one color and another region to be lit ambiently with another color, you could use an ambient light with location and falloff. You can think of it as a point light that doesn't care which direction a surface is facing.

Syntax `lightFalloff(constant, linear, quadratic)`

Parameters
constant float: constant value or determining falloff
linear float: linear value for determining falloff
quadratic float: quadratic value for determining falloff

Returns void

Related lights()
ambientLight()
pointLight()
spotLight()
lightSpecular()

Name

lights()

```
size(100, 100, P3D);
background(0);
noStroke();
// Sets the default ambient
// and directional light
lights();
translate(20, 50, 0);
sphere(30);
translate(60, 0, 0);
sphere(30);
```

Examples

```
void setup() {
  size(100, 100, P3D);
  background(0);
  noStroke();
}

void draw() {
  // Include lights() at the beginning
  // of draw() to keep them persistent
  lights();
  translate(20, 50, 0);
  sphere(30);
  translate(60, 0, 0);
  sphere(30);
}
```

Description Sets the default ambient light, directional light, falloff, and specular values. The defaults are ambientLight(128, 128, 128) and directionalLight(128, 128, 128, 0, 0, -1), lightFalloff(1, 0, 0), and lightSpecular(0, 0, 0). Lights need to be included in the draw() to remain persistent in a looping program. Placing them in the setup() of a looping program will cause them to only have an effect the first time through the loop.

Syntax `lights()`

Returns void

Related
ambientLight()
directionalLight()
pointLight()
spotLight()
noLights()

Name

lightSpecular()

Examples

```
size(100, 100, P3D);
background(0);
noStroke();
directionalLight(102, 102, 102, 0, 0, -1);
```

```
lightSpecular(204, 204, 204);
directionalLight(102, 102, 102, 0, 1, -1);
lightSpecular(102, 102, 102);
translate(20, 50, 0);
specular(51, 51, 51);
sphere(30);
translate(60, 0, 0);
specular(102, 102, 102);
sphere(30);
```

Description Sets the specular color for lights. Like **fill()**, it affects only the elements which are created after it in the code. Specular refers to light which bounces off a surface in a preferred direction (rather than bouncing in all directions like a diffuse light) and is used for creating highlights. The specular quality of a light interacts with the specular material qualities set through the **specular()** and **shininess()** functions.

Syntax `lightSpecular(v1, v2, v3)`

Parameters
v1 float: red or hue value (depending on current color mode)
v2 float: green or saturation value (depending on current color mode)
v3 float: blue or brightness value (depending on current color mode)

Returns void

Related specular()
lights()
ambientLight()
pointLight()
spotLight()

Name

noLights()

Description Disable all lighting. Lighting is turned off by default and enabled with the **lights()** function. This function can be used to disable lighting so that 2D geometry (which does not require lighting) can be drawn after a set of lighted 3D geometry.

Syntax `noLights()`

Returns void

Related lights()

Name

normal()

Examples

```
size(100, 100, P3D);
noStroke();
background(0);
pointLight(150, 250, 150, 10, 30, 50);
beginShape();
normal(0, 0, 1);
vertex(20, 20, -10);
vertex(80, 20, 10);
vertex(80, 80, -10);
vertex(20, 80, 10);
endShape(CLOSE);
```

Description Sets the current normal vector. Used for drawing three dimensional shapes and surfaces, **normal()** specifies a vector perpendicular to a shape's surface which, in turn, determines how lighting affects it. Processing attempts to automatically assign normals to shapes, but since that's imperfect, this is a better option when you want more control. This function is identical to **glNormal3f()** in OpenGL.

Syntax
```
normal(nx, ny, nz)
```

Parameters
nx float: x direction
ny float: y direction
nz float: z direction

Returns void

Related
beginShape()
endShape()
lights()

Name

pointLight()

Examples
```
size(100, 100, P3D);
background(0);
noStroke();
pointLight(51, 102, 126, 35, 40, 36);
translate(80, 50, 0);
sphere(30);
```

Description Adds a point light. Lights need to be included in the **draw()** to remain persistent in a looping program. Placing them in the **setup()** of a looping program will cause them to only have an effect the first time through the loop. The **v1**, **v2**, and **v3** parameters are interpreted as either RGB or HSB values, depending on the current color mode. The **x**, **y**, and **z** parameters set the position of the light.

Syntax
```
pointLight(v1, v2, v3, x, y, z)
```

Parameters
v1 float: red or hue value (depending on current color mode)
v2 float: green or saturation value (depending on current color mode)
v3 float: blue or brightness value (depending on current color mode)
x float: x-coordinate of the light
y float: y-coordinate of the light
z float: z-coordinate of the light

Returns void

Related
lights()
directionalLight()
ambientLight()
spotLight()

Name

spotLight()

Examples
```
size(100, 100, P3D);
background(0);
```

```
noStroke();
spotLight(51, 102, 126, 80, 20, 40, -1, 0, 0, PI/2, 2);
translate(20, 50, 0);
sphere(30);


size(100, 100, P3D);
int concentration = 600;  // Try 1 -> 10000
background(0);
noStroke();
spotLight(51, 102, 126, 50, 50, 400,
          0, 0, -1, PI/16, concentration);
translate(80, 50, 0);
sphere(30);
```

Description

Adds a spot light. Lights need to be included in the **draw()** to remain persistent in a looping program. Placing them in the **setup()** of a looping program will cause them to only have an effect the first time through the loop. The **v1**, **v2**, and **v3** parameters are interpreted as either RGB or HSB values, depending on the current color mode. The **x**, **y**, and **z** parameters specify the position of the light and **nx**, **ny**, **nz** specify the direction of light. The **angle** parameter affects angle of the spotlight cone, while **concentration** sets the bias of light focusing toward the center of that cone.

Syntax

```
spotLight(v1, v2, v3, x, y, z, nx, ny, nz, angle, concentration)
```

Parameters

v1	float: red or hue value (depending on current color mode)
v2	float: green or saturation value (depending on current color mode)
v3	float: blue or brightness value (depending on current color mode)
x	float: x-coordinate of the light
y	float: y-coordinate of the light
z	float: z-coordinate of the light
nx	float: direction along the x axis
ny	float: direction along the y axis
nz	float: direction along the z axis
angle	float: angle of the spotlight cone
concentration	float: exponent determining the center bias of the cone

Returns

void

Related

lights()
directionalLight()
pointLight()
ambientLight()

Name

beginCamera()

Examples

```
size(100, 100, P3D);
noFill();

beginCamera();
camera();
rotateX(-PI/6);
endCamera();
```

```
translate(50, 50, 0);
rotateY(PI/3);
box(45);
```

Description The **beginCamera()** and **endCamera()** functions enable advanced customization of the camera space. The functions are useful if you want to more control over camera movement, however for most users, the **camera()** function will be sufficient.

The camera functions will replace any transformations (such as **rotate()** or **translate()**) that occur before them in **draw()**, but they will not automatically replace the camera transform itself. For this reason, camera functions should be placed at the beginning of **draw()** (so that transformations happen afterwards), and the **camera()** function can be used after **beginCamera()** if you want to reset the camera before applying transformations.

This function sets the matrix mode to the camera matrix so calls such as **translate()**, **rotate()**, applyMatrix() and resetMatrix() affect the camera. **beginCamera()** should always be used with a following **endCamera()** and pairs of **beginCamera()** and **endCamera()** cannot be nested.

Syntax
```
beginCamera()
```

Returns void

Related
camera()
endCamera()
applyMatrix()
resetMatrix()
translate()
scale()

Name

camera()

Examples
```
size(100, 100, P3D);
noFill();
background(204);
camera(70.0, 35.0, 120.0, 50.0, 50.0, 0.0,
       0.0, 1.0, 0.0);
translate(50, 50, 0);
rotateX(-PI/6);
rotateY(PI/3);
box(45);
```

Description Sets the position of the camera through setting the eye position, the center of the scene, and which axis is facing upward. Moving the eye position and the direction it is pointing (the center of the scene) allows the images to be seen from different angles. The version without any parameters sets the camera to the default position, pointing to the center of the display window with the Y axis as up. The default values are **camera(width/2.0, height/2.0, (height/2.0) / tan(PI*30.0 / 180.0), width/2.0, height/2.0, 0, 0, 1, 0)**. This function is similar to **gluLookAt()** in OpenGL, but it first clears the current camera settings.

Syntax
```
camera()
camera(eyeX, eyeY, eyeZ, centerX, centerY, centerZ, upX, upY,
upZ)
```

	eyeX	float: x-coordinate for the eye
	eyeY	float: y-coordinate for the eye
	eyeZ	float: z-coordinate for the eye
	centerX	float: x-coordinate for the center of the scene
Parameters	**centerY**	float: y-coordinate for the center of the scene
	centerZ	float: z-coordinate for the center of the scene
	upX	float: usually 0.0, 1.0, or -1.0
	upY	float: usually 0.0, 1.0, or -1.0
	upZ	float: usually 0.0, 1.0, or -1.0
Returns	void	
Related	endCamera() frustum()	

Name

endCamera()

Examples

```
size(100, 100, P3D);
noFill();

beginCamera();
camera();
rotateX(-PI/6);
endCamera();

translate(50, 50, 0);
rotateY(PI/3);
box(45);
```

Description The **beginCamera()** and **endCamera()** functions enable advanced customization of the camera space. Please see the reference for **beginCamera()** for a description of how the functions are used.

Syntax `endCamera()`

Returns void

Related camera()

Name

frustum()

Examples

```
size(100, 100, P3D);
noFill();
background(204);
frustum(-10, 0, 0, 10, 10, 200);
rotateY(PI/6);
box(45);
```

Sets a perspective matrix as defined by the parameters.

Description A frustum is a geometric form: a pyramid with its top cut off. With the viewer's eye at the imaginary top of the pyramid, the six planes of the frustum act as clipping planes when rendering a 3D view. Thus, any form inside the clipping planes is rendered and visible; anything outside those planes is not visible.

Setting the frustum has the effect of changing the *perspective* with which the scene is rendered. This can be acheived more simply in many cases by using **perspective()**.

Note that the near value must be greater than zero (as the point of the frustum "pyramid" cannot converge "behind" the viewer). Similarly, the far value must be greater than the near value (as the "far" plane of the frustum must be "farther away" from the viewer than the near plane).

Works like glFrustum, except it wipes out the current perspective matrix rather than multiplying itself with it.

Syntax

```
frustum(left, right, bottom, top, near, far)
```

Parameters

left	float: left coordinate of the clipping plane
right	float: right coordinate of the clipping plane
bottom	float: bottom coordinate of the clipping plane
top	float: top coordinate of the clipping plane
near	float: near component of the clipping plane; must be greater than zero
far	float: far component of the clipping plane; must be greater than the near value

Returns void

Related camera()
endCamera()
perspective()

Name

ortho()

Examples

```
size(100, 100, P3D);
noFill();
ortho(0, width, 0, height); // same as ortho()
translate(width/2, height/2, 0);
rotateX(-PI/6);
rotateY(PI/3);
box(45);
```

Description

Sets an orthographic projection and defines a parallel clipping volume. All objects with the same dimension appear the same size, regardless of whether they are near or far from the camera. The parameters to this function specify the clipping volume where left and right are the minimum and maximum x values, top and bottom are the minimum and maximum y values, and near and far are the minimum and maximum z values. If no parameters are given, the default is used: ortho(0, width, 0, height).

Syntax

```
ortho()
ortho(left, right, bottom, top)
ortho(left, right, bottom, top, near, far)
```

Parameters

left	float: left plane of the clipping volume
right	float: right plane of the clipping volume
bottom	float: bottom plane of the clipping volume
top	float: top plane of the clipping volume
near	float: maximum distance from the origin to the viewer

far float: maximum distance from the origin away from the viewer

Returns void

Name

perspective()

Examples

```
// Re-creates the default perspective
size(100, 100, P3D);
noFill();
float fov = PI/3.0;
float cameraZ = (height/2.0) / tan(fov/2.0);
perspective(fov, float(width)/float(height),
            cameraZ/10.0, cameraZ*10.0);
translate(50, 50, 0);
rotateX(-PI/6);
rotateY(PI/3);
box(45);
```

Description Sets a perspective projection applying foreshortening, making distant objects appear smaller than closer ones. The parameters define a viewing volume with the shape of truncated pyramid. Objects near to the front of the volume appear their actual size, while farther objects appear smaller. This projection simulates the perspective of the world more accurately than orthographic projection. The version of perspective without parameters sets the default perspective and the version with four parameters allows the programmer to set the area precisely. The default values are: perspective(PI/3.0, width/height, cameraZ/10.0, cameraZ*10.0) where cameraZ is ((height/2.0) / tan(PI*60.0/360.0));

Syntax

```
perspective()
perspective(fovy, aspect, zNear, zFar)
```

Parameters

fovy float: field-of-view angle (in radians) for vertical direction

aspect float: ratio of width to height

zNear float: z-position of nearest clipping plane

zFar float: z-position of farthest clipping plane

Returns void

Name

printCamera()

Examples

```
size(100, 100, P3D);
printCamera();

// The program above prints this data:
// 01.0000  00.0000  00.0000 -50.0000
// 00.0000  01.0000  00.0000 -50.0000
// 00.0000  00.0000  01.0000 -86.6025
// 00.0000  00.0000  00.0000  01.0000
```

Description Prints the current camera matrix to the Console (the text window at the bottom of Processing).

Syntax

```
printCamera()
```

Returns void

Related camera()

Name

printProjection()

Examples

```
size(100, 100, P3D);
printProjection();

// The program above prints this data:
// 01.7321  00.0000  00.0000  00.0000
// 00.0000 -01.7321  00.0000  00.0000
// 00.0000  00.0000 -01.0202 -17.4955
// 00.0000  00.0000 -01.0000  00.0000
```

Description Prints the current projection matrix to the Console (the text window at the bottom of Processing).

Syntax `printProjection()`

Returns void

Related camera()

Name

modelX()

Examples

```
void setup() {
  size(500, 500, P3D);
  noFill();
}

void draw() {
  background(0);

  pushMatrix();
  // start at the middle of the screen
  translate(width/2, height/2, -200);
  // some random rotation to make things interesting
  rotateY(1.0); //yrot);
  rotateZ(2.0); //zrot);
  // rotate in X a little more each frame
  rotateX(frameCount / 100.0);
  // offset from center
  translate(0, 150, 0);

  // draw a white box outline at (0, 0, 0)
  stroke(255);
  box(50);

  // the box was drawn at (0, 0, 0), store that location
  float x = modelX(0, 0, 0);
  float y = modelY(0, 0, 0);
  float z = modelZ(0, 0, 0);
  // clear out all the transformations
  popMatrix();

  // draw another box at the same (x, y, z) coordinate as the
other
  pushMatrix();
  translate(x, y, z);
  stroke(255, 0, 0);
  box(50);
  popMatrix();
```

```
}
```

Returns the three-dimensional X, Y, Z position in model space. This returns the X value for a given coordinate based on the current set of transformations (scale, rotate, translate, etc.) The X value can be used to place an object in space relative to the location of the original point once the transformations are no longer in use.

Description In the example, the **modelX()**, **modelY()**, and **modelZ()** functions record the location of a box in space after being placed using a series of translate and rotate commands. After popMatrix() is called, those transformations no longer apply, but the (x, y, z) coordinate returned by the model functions is used to place another box in the same location.

Syntax `modelX(x, y, z)`

Parameters
x float: 3D x-coordinate to be mapped
y float: 3D y-coordinate to be mapped
z float: 3D z-coordinate to be mapped

Returns float

Related modelY()
modelZ()

Name

modelY()

Examples

```
void setup() {
  size(500, 500, P3D);
  noFill();
}

void draw() {
  background(0);

  pushMatrix();
  // start at the middle of the screen
  translate(width/2, height/2, -200);
  // some random rotation to make things interesting
  rotateY(1.0); //yrot);
  rotateZ(2.0); //zrot);
  // rotate in X a little more each frame
  rotateX(frameCount / 100.0);
  // offset from center
  translate(0, 150, 0);

  // draw a white box outline at (0, 0, 0)
  stroke(255);
  box(50);

  // the box was drawn at (0, 0, 0), store that location
  float x = modelX(0, 0, 0);
  float y = modelY(0, 0, 0);
  float z = modelZ(0, 0, 0);
  // clear out all the transformations
  popMatrix();

  // draw another box at the same (x, y, z) coordinate as the
other
  pushMatrix();
  translate(x, y, z);
```

```
  stroke(255, 0, 0);
  box(50);
  popMatrix();
}
```

Returns the three-dimensional X, Y, Z position in model space. This returns the Y value for a given coordinate based on the current set of transformations (scale, rotate, translate, etc.) The Y value can be used to place an object in space relative to the location of the original point once the transformations are no longer in use.

Description In the example, the **modelX()**, **modelY()**, and **modelZ()** functions record the location of a box in space after being placed using a series of translate and rotate commands. After popMatrix() is called, those transformations no longer apply, but the (x, y, z) coordinate returned by the model functions is used to place another box in the same location.

Syntax `modelY(x, y, z)`

Parameters
x float: 3D x-coordinate to be mapped
y float: 3D y-coordinate to be mapped
z float: 3D z-coordinate to be mapped

Returns float

Related modelX()
modelZ()

Name

modelZ()

Examples

```
void setup() {
  size(500, 500, P3D);
  noFill();
}

void draw() {
  background(0);

  pushMatrix();
  // start at the middle of the screen
  translate(width/2, height/2, -200);
  // some random rotation to make things interesting
  rotateY(1.0); //yrot);
  rotateZ(2.0); //zrot);
  // rotate in X a little more each frame
  rotateX(frameCount / 100.0);
  // offset from center
  translate(0, 150, 0);

  // draw a white box outline at (0, 0, 0)
  stroke(255);
  box(50);

  // the box was drawn at (0, 0, 0), store that location
  float x = modelX(0, 0, 0);
  float y = modelY(0, 0, 0);
  float z = modelZ(0, 0, 0);
  // clear out all the transformations
  popMatrix();

  // draw another box at the same (x, y, z) coordinate as the
```

```
other
  pushMatrix();
  translate(x, y, z);
  stroke(255, 0, 0);
  box(50);
  popMatrix();
}
```

Returns the three-dimensional X, Y, Z position in model space. This returns the Z value for a given coordinate based on the current set of transformations (scale, rotate, translate, etc.) The Z value can be used to place an object in space relative to the location of the original point once the transformations are no longer in use.

Description In the example, the **modelX()**, **modelY()**, and **modelZ()** functions record the location of a box in space after being placed using a series of translate and rotate commands. After popMatrix() is called, those transformations no longer apply, but the (x, y, z) coordinate returned by the model functions is used to place another box in the same location.

Syntax `modelZ(x, y, z)`

Parameters
x float: 3D x-coordinate to be mapped
y float: 3D y-coordinate to be mapped
z float: 3D z-coordinate to be mapped

Returns float

Related modelX()
modelY()

Name

screenX()

Examples

```
void setup() {
  size(100, 100, P3D);
}

void draw() {
  background(204);

  float x = mouseX;
  float y = mouseY;
  float z = -100;

  // Draw "X" at z = -100
  stroke(255);
  line(x-10, y-10, z, x+10, y+10, z);
  line(x+10, y-10, z, x-10, y+10, z);

  // Draw gray line at z = 0 and same
  // x value. Notice the parallax
  stroke(102);
  line(x, 0, 0, x, height, 0);

  // Draw black line at z = 0 to match
  // the x value element drawn at z = -100
  stroke(0);
  float theX = screenX(x, y, z);
  line(theX, 0, 0, theX, height, 0);
}
```

Description Takes a three-dimensional X, Y, Z position and returns the X value for where it

will appear on a (two-dimensional) screen.

Syntax
```
screenX(x, y)
screenX(x, y, z)
```

Parameters
x float: 3D x-coordinate to be mapped
y float: 3D y-coordinate to be mapped
z float: 3D z-coordinate to be mapped

Returns float

Related screenY()
screenZ()

Name

screenY()

Examples
```
void setup() {
  size(100, 100, P3D);
}

void draw() {
  background(204);

  float x = mouseX;
  float y = mouseY;
  float z = -100;

  // Draw "X" at z = -100
  stroke(255);
  line(x-10, y-10, z, x+10, y+10, z);
  line(x+10, y-10, z, x-10, y+10, z);

  // Draw gray line at z = 0 and same
  // y value. Notice the parallax
  stroke(102);
  line(0, y, 0, width, y, 0);

  // Draw black line at z = 0 to match
  // the y value element drawn at z = -100
  stroke(0);
  float theY = screenY(x, y, z);
  line(0, theY, 0, width, theY, 0);
}
```

Description Takes a three-dimensional X, Y, Z position and returns the Y value for where it will appear on a (two-dimensional) screen.

Syntax
```
screenY(x, y)
screenY(x, y, z)
```

Parameters
x float: 3D x-coordinate to be mapped
y float: 3D y-coordinate to be mapped
z float: 3D z-coordinate to be mapped

Returns float

Related screenX()
screenZ()

Name

screenZ()

Examples
```
Coming soon...
```

Description Takes a three-dimensional X, Y, Z position and returns the Z value for where it will appear on a (two-dimensional) screen.

Syntax
```
screenZ(x, y, z)
```

Parameters
x float: 3D x-coordinate to be mapped
y float: 3D y-coordinate to be mapped
z float: 3D z-coordinate to be mapped

Returns float

Related screenX()
screenY()

Name

ambient()

Examples
```
size(100, 100, P3D);
background(0);
noStroke();
directionalLight(153, 153, 153, .5, 0, -1);
ambientLight(153, 102, 0);
ambient(51, 26, 0);
translate(70, 50, 0);
sphere(30);
```

Description Sets the ambient reflectance for shapes drawn to the screen. This is combined with the ambient light component of environment. The color components set through the parameters define the reflectance. For example in the default color mode, setting v1=255, v2=127, v3=0, would cause all the red light to reflect and half of the green light to reflect. Used in combination with **emissive()**, **specular()**, and **shininess()** in setting the material properties of shapes.

Syntax
```
ambient(rgb)
ambient(gray)
ambient(v1, v2, v3)
```

Parameters
rgb int: any value of the color datatype
gray float: number specifying value between white and black
v1 float: red or hue value (depending on current color mode)
v2 float: green or saturation value (depending on current color mode)
v3 float: blue or brightness value (depending on current color mode)

Returns void

Related emissive()
specular()
shininess()

Name

emissive()

Examples
```
size(100, 100, P3D);
background(0);
noStroke();
background(0);
```

```
directionalLight(204, 204, 204, .5, 0, -1);
emissive(0, 26, 51);
translate(70, 50, 0);
sphere(30);
```

Description Sets the emissive color of the material used for drawing shapes drawn to the screen. Used in combination with **ambient()**, **specular()**, and **shininess()** in setting the material properties of shapes.

Syntax
```
emissive(rgb)
emissive(gray)
emissive(v1, v2, v3)
```

Parameters
rgb int: color to set
v1 float: red or hue value (depending on current color mode)
v2 float: green or saturation value (depending on current color mode)
v3 float: blue or brightness value (depending on current color mode)

Returns void

Related ambient()
specular()
shininess()

Name

shininess()

```
size(100, 100, P3D);
background(0);
noStroke();
background(0);
fill(0, 51, 102);
ambientLight(102, 102, 102);
lightSpecular(204, 204, 204);
directionalLight(102, 102, 102, 0, 0, -1);
specular(255, 255, 255);
translate(30, 50, 0);
shininess(1.0);
sphere(20);  // Left sphere
translate(40, 0, 0);
shininess(5.0);
sphere(20);  // Right sphere
```

Description Sets the amount of gloss in the surface of shapes. Used in combination with **ambient()**, **specular()**, and **emissive()** in setting the material properties of shapes.

Syntax
```
shininess(shine)
```

Parameters **shine** float: degree of shininess

Returns void

Related emissive()
ambient()
specular()

Name

specular()

Examples

```
size(100, 100, P3D);
background(0);
noStroke();
background(0);
fill(0, 51, 102);
lightSpecular(255, 255, 255);
directionalLight(204, 204, 204, 0, 0, -1);
translate(20, 50, 0);
specular(255, 255, 255);
sphere(30);
translate(60, 0, 0);
specular(204, 102, 0);
sphere(30);
```

Description Sets the specular color of the materials used for shapes drawn to the screen, which sets the color of highlights. Specular refers to light which bounces off a surface in a preferred direction (rather than bouncing in all directions like a diffuse light). Used in combination with **emissive()**, **ambient()**, and **shininess()** in setting the material properties of shapes.

Syntax
```
specular(rgb)
specular(gray)
specular(v1, v2, v3)
```

Parameters
- **rgb** int: color to set
- **v1** float: red or hue value (depending on current color mode)
- **v2** float: green or saturation value (depending on current color mode)
- **v3** float: blue or brightness value (depending on current color mode)

Returns void

Related
lightSpecular()
ambient()
emissive()
shininess()

Name

background()

Examples
```
background(51);
```

```
background(255, 204, 0);
```

```
PImage img;
img = loadImage("laDefense.jpg");
background(img);
```

Description The **background()** function sets the color used for the background of the Processing window. The default background is light gray. This function is typically used within **draw()** to clear the display window at the beginning of each frame, but it can be used inside **setup()** to set the background on the first frame of animation or if the backgound need only be set once.

An image can also be used as the background for a sketch, although the image's width and height must match that of the sketch window. Images used with **background()** will ignore the current **tint()** setting. To resize an image to the size

of the sketch window, use image.resize(width, height).

It is not possible to use the transparency **alpha** parameter with background colors on the main drawing surface. It can only be used along with a **PGraphics** object and **createGraphics()**.

Syntax

```
background(rgb)
background(rgb, alpha)
background(gray)
background(gray, alpha)
background(v1, v2, v3)
background(v1, v2, v3, alpha)
background(image)
```

Parameters

rgb	int: any value of the color datatype
alpha	float: opacity of the background
gray	float: specifies a value between white and black
v1	float: red or hue value (depending on the current color mode)
v2	float: green or saturation value (depending on the current color mode)
v3	float: blue or brightness value (depending on the current color mode)
image	PImage: PImage to set as background (must be same size as the sketch window)

Returns void

Related stroke() fill() tint() colorMode()

Name

clear()

Examples

```
PGraphics pg;

void setup() {
  size(200, 200);
  pg = createGraphics(100, 100);
}

void draw() {
  background(204);
  pg.beginDraw();
  pg.stroke(0, 102, 153);
  pg.line(0, 0, mouseX, mouseY);
  pg.endDraw();
  image(pg, 50, 50);
}

// Click to clear the PGraphics object
void mousePressed() {
  pg.clear();
}
```

Description Clears the pixels within a buffer. This function only works on PGraphics objects created with the **createGraphics()** function; it won't work with the main display window. Unlike the main graphics context (the display window), pixels in additional graphics areas created with **createGraphics()** can be entirely or partially transparent. This function clears everything to make all of the pixels 100% transparent.

Syntax `clear()`
Returns void

Name

colorMode()

Examples

```
noStroke();
colorMode(RGB, 100);
for (int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    stroke(i, j, 0);
    point(i, j);
  }
}
```

```
noStroke();
colorMode(HSB, 100);
for (int i = 0; i < 100; i++) {
  for (int j = 0; j < 100; j++) {
    stroke(i, j, 100);
    point(i, j);
  }
}
```

```
// If the color is defined here, it won't be
// affected by the colorMode() in setup().
// Instead, just declare the variable here and
// assign the value after the colorMode() in setup()
//color bg = color(180, 50, 50);  // No
color bg;  // Yes, but assign it in setup()

void setup() {
  size(100, 100);
  colorMode(HSB, 360, 100, 100);
  bg = color(180, 50, 50);
}

void draw() {
  background(bg);
}
```

Description Changes the way Processing interprets color data. By default, the parameters for **fill()**, **stroke()**, **background()**, and **color()** are defined by values between 0 and 255 using the RGB color model. The **colorMode()** function is used to change the numerical range used for specifying colors and to switch color systems. For example, calling **colorMode(RGB, 1.0)** will specify that values are specified between 0 and 1. The limits for defining colors are altered by setting the parameters range1, range2, range3, and range 4.

Syntax

```
colorMode(mode)
colorMode(mode, max)
colorMode(mode, max1, max2, max3)
colorMode(mode, max1, max2, max3, maxA)
```

Parameters **mode** int: Either RGB or HSB, corresponding to Red/Green/Blue and Hue/Saturation/Brightness

max float: range for all color elements
max1 float: range for the red or hue depending on the current color mode
max2 float: range for the green or saturation depending on the current color mode
max3 float: range for the blue or brightness depending on the current color mode
maxA float: range for the alpha

Returns void

Related background()
fill()
stroke()

Name

fill()

Examples

```
fill(153);
rect(30, 20, 55, 55);
```

```
fill(204, 102, 0);
rect(30, 20, 55, 55);
```

Description

Sets the color used to fill shapes. For example, if you run **fill(204, 102, 0)**, all subsequent shapes will be filled with orange. This color is either specified in terms of the RGB or HSB color depending on the current **colorMode()**. (The default color space is RGB, with each value in the range from 0 to 255.)

When using hexadecimal notation to specify a color, use "#" or "0x" before the values (e.g., #CCFFAA or 0xFFCCFFAA). The # syntax uses six digits to specify a color (just as colors are typically specified in HTML and CSS). When using the hexadecimal notation starting with "0x", the hexadecimal value must be specified with eight characters; the first two characters define the alpha component, and the remainder define the red, green, and blue components.

The value for the "gray" parameter must be less than or equal to the current maximum value as specified by **colorMode()**. The default maximum value is 255.

To change the color of an image or a texture, use **tint()**.

Syntax

```
fill(rgb)
fill(rgb, alpha)
fill(gray)
fill(gray, alpha)
fill(v1, v2, v3)
fill(v1, v2, v3, alpha)
```

Parameters

rgb int: color variable or hex value
alpha float: opacity of the fill
gray float: number specifying value between white and black
v1 float: red or hue value (depending on current color mode)
v2 float: green or saturation value (depending on current color mode)
v3 float: blue or brightness value (depending on current color mode)

Returns void

Related
noFill()
stroke()
tint()
background()
colorMode()

Name

noFill()

Examples

```
rect(15, 10, 55, 55);
noFill();
rect(30, 20, 55, 55);
```

Description Disables filling geometry. If both **noStroke()** and **noFill()** are called, nothing will be drawn to the screen.

Syntax

```
noFill()
```

Returns void

Related fill()

Name

noStroke()

Examples

```
noStroke();
rect(30, 20, 55, 55);
```

Description Disables drawing the stroke (outline). If both **noStroke()** and **noFill()** are called, nothing will be drawn to the screen.

Syntax

```
noStroke()
```

Returns void

Related stroke()

Name

stroke()

Examples

```
stroke(153);
rect(30, 20, 55, 55);
```

```
stroke(204, 102, 0);
rect(30, 20, 55, 55);
```

Description Sets the color used to draw lines and borders around shapes. This color is either specified in terms of the RGB or HSB color depending on the current **colorMode()** (the default color space is RGB, with each value in the range from 0 to 255).

When using hexadecimal notation to specify a color, use "#" or "0x" before the values (e.g. #CCFFAA, 0xFFCCFFAA). The # syntax uses six digits to specify a

color (the way colors are specified in HTML and CSS). When using the hexadecimal notation starting with "0x", the hexadecimal value must be specified with eight characters; the first two characters define the alpha component and the remainder the red, green, and blue components.

The value for the gray parameter must be less than or equal to the current maximum value as specified by **colorMode()**. The default maximum value is 255.

When drawing in 2D with the default renderer, you may need **hint(ENABLE_STROKE_PURE)** to improve drawing quality (at the expense of performance). See the hint() documentation for more details.

Syntax

```
stroke(rgb)
stroke(rgb, alpha)
stroke(gray)
stroke(gray, alpha)
stroke(v1, v2, v3)
stroke(v1, v2, v3, alpha)
```

Parameters

rgb	int: color value in hexadecimal notation
alpha	float: opacity of the stroke
gray	float: specifies a value between white and black
v1	float: red or hue value (depending on current color mode)
v2	float: green or saturation value (depending on current color mode)
v3	float: blue or brightness value (depending on current color mode)

Returns void

Name

alpha()

Examples

```
noStroke();
color c = color(0, 126, 255, 102);
fill(c);
rect(15, 15, 35, 70);
float value = alpha(c);  // Sets 'value' to 102
fill(value);
rect(50, 15, 35, 70);
```

Description Extracts the alpha value from a color.

Syntax

```
alpha(rgb)
```

Parameters **rgb** int: any value of the color datatype

Returns float

Related

red()
green()
blue()
hue()
saturation()
brightness()

Name

blue()

Examples

```
color c = color(175, 100, 220);  // Define color 'c'
fill(c);  // Use color variable 'c' as fill color
rect(15, 20, 35, 60);  // Draw left rectangle

float blueValue = blue(c);  // Get blue in 'c'
println(blueValue);  // Prints "220.0"
fill(0, 0, blueValue);  // Use 'blueValue' in new fill
rect(50, 20, 35, 60);  // Draw right rectangle
```

Description Extracts the blue value from a color, scaled to match current **colorMode()**. The value is always returned as a float, so be careful not to assign it to an int value.

The **blue()** function is easy to use and understand, but it is slower than a technique called bit masking. When working in **colorMode(RGB, 255)**, you can acheive the same results as **blue()** but with greater speed by using a bit mask to remove the other color components. For example, the following two lines of code are equivalent means of getting the blue value of the color value **c**:

```
float b1 = blue(c);   // Simpler, but slower to calculate
float b2 = c & 0xFF;  // Very fast to calculate
```

Syntax `blue(rgb)`

Parameters **rgb** int: any value of the color datatype

Returns float

Related red()
green()
alpha()
hue()
saturation()
brightness()
>> (right shift)

Name

brightness()

Examples

```
noStroke();
colorMode(HSB, 255);
color c = color(0, 126, 255);
fill(c);
rect(15, 20, 35, 60);
float value = brightness(c);  // Sets 'value' to 255
fill(value);
rect(50, 20, 35, 60);
```

Description Extracts the brightness value from a color.

Syntax `brightness(rgb)`

Parameters **rgb** int: any value of the color datatype

Returns float

Related red()
green()
blue()
alpha()

hue()
saturation()

Name

color()

Examples

```
color c = color(255, 204, 0);  // Define color 'c'
fill(c);  // Use color variable 'c' as fill color
noStroke();  // Don't draw a stroke around shapes
rect(30, 20, 55, 55);  // Draw rectangle
```

```
color c = color(255, 204, 0);  // Define color 'c'
fill(c);  // Use color variable 'c' as fill color
noStroke();  // Don't draw a stroke around shapes
ellipse(25, 25, 80, 80);  // Draw left circle

// Using only one value with color()
// generates a grayscale value.
c = color(65);  // Update 'c' with grayscale value
fill(c);  // Use updated 'c' as fill color
ellipse(75, 75, 80, 80);  // Draw right circle
```

```
color c;  // Declare color 'c'
noStroke();  // Don't draw a stroke around shapes

// If no colorMode is specified, then the
// default of RGB with scale of 0-255 is used.
c = color(50, 55, 100);  // Create a color for 'c'
fill(c);  // Use color variable 'c' as fill color
rect(0, 10, 45, 80);  // Draw left rect

colorMode(HSB, 100);  // Use HSB with scale of 0-100
c = color(50, 55, 100);  // Update 'c' with new color
fill(c);  // Use updated 'c' as fill color
rect(55, 10, 45, 80);  // Draw right rect
```

Description

Creates colors for storing in variables of the **color** datatype. The parameters are interpreted as RGB or HSB values depending on the current **colorMode()**. The default mode is RGB values from 0 to 255 and, therefore, the function call **color(255, 204, 0)** will return a bright yellow color (see the first example above).

Note that if only one value is provided to **color()**, it will be interpreted as a grayscale value. Add a second value, and it will be used for alpha transparency. When three values are specified, they are interpreted as either RGB or HSB values. Adding a fourth value applies alpha transparency.

More about how colors are stored can be found in the reference for the color datatype.

Syntax

```
color(gray)
color(gray, alpha)
color(v1, v2, v3)
color(v1, v2, v3, alpha)
```

Parameters **gray** int: number specifying value between white and black

alpha float, or int: relative to current color range
v1 float, or int: red or hue values relative to the current color range
v2 float, or int: green or saturation values relative to the current color range
v3 float, or int: blue or brightness values relative to the current color range

Returns int

Related colorMode()

Name

green()

Examples

```
color c = color(20, 75, 200);  // Define color 'c'
fill(c);  // Use color variable 'c' as fill color
rect(15, 20, 35, 60);  // Draw left rectangle

float greenValue = green(c);  // Get green in 'c'
println(greenValue);  // Print "75.0"
fill(0, greenValue, 0);  // Use 'greenValue' in new fill
rect(50, 20, 35, 60);  // Draw right rectangle
```

Description Extracts the green value from a color, scaled to match current **colorMode()**. The value is always returned as a float, so be careful not to assign it to an int value.

The **green()** function is easy to use and understand, but it is slower than a technique called bit shifting. When working in **colorMode(RGB, 255)**, you can acheive the same results as **green()** but with greater speed by using the right shift operator (>>) with a bit mask. For example, the following two lines of code are equivalent means of getting the green value of the color value **c**:

```
float r1 = green(c);  // Simpler, but slower to calculate
float r2 = c >> 8 & 0xFF;  // Very fast to calculate
```

Syntax `green(rgb)`

Parameters **rgb** int: any value of the color datatype

Returns float

Related red()
blue()
alpha()
hue()
saturation()
brightness()
>> (right shift)

Name

hue()

Examples

```
noStroke();
colorMode(HSB, 255);
color c = color(0, 126, 255);
fill(c);
rect(15, 20, 35, 60);
float value = hue(c);  // Sets 'value' to "0"
```

```
fill(value);
rect(50, 20, 35, 60);
```

Description Extracts the hue value from a color.

Syntax `hue(rgb)`

Parameters **rgb** int: any value of the color datatype

Returns float

Related red()
green()
blue()
alpha()
saturation()
brightness()

Name

lerpColor()

Examples

```
stroke(255);
background(51);
color from = color(204, 102, 0);
color to = color(0, 102, 153);
color interA = lerpColor(from, to, .33);
color interB = lerpColor(from, to, .66);
fill(from);
rect(10, 20, 20, 60);
fill(interA);
rect(30, 20, 20, 60);
fill(interB);
rect(50, 20, 20, 60);
fill(to);
rect(70, 20, 20, 60);
```

Description Calculates a color or colors between two color at a specific increment. The **amt** parameter is the amount to interpolate between the two values where 0.0 equal to the first point, 0.1 is very near the first point, 0.5 is halfway in between, etc.

Syntax `lerpColor(c1, c2, amt)`

Parameters **c1** int: interpolate from this color
c2 int: interpolate to this color
amt float: between 0.0 and 1.0

Returns int

Related color()

Name

red()

Examples

```
color c = color(255, 204, 0);  // Define color 'c'
fill(c);  // Use color variable 'c' as fill color
rect(15, 20, 35, 60);  // Draw left rectangle

float redValue = red(c);  // Get red in 'c'
println(redValue);  // Print "255.0"
fill(redValue, 0, 0);  // Use 'redValue' in new fill
```

```
rect(50, 20, 35, 60);  // Draw right rectangle
```

Description Extracts the red value from a color, scaled to match current **colorMode()**. The value is always returned as a float, so be careful not to assign it to an int value.

The **red()** function is easy to use and understand, but it is slower than a technique called bit shifting. When working in **colorMode(RGB, 255)**, you can acheive the same results as **red()** but with greater speed by using the right shift operator (>>) with a bit mask. For example, the following two lines of code are equivalent means of getting the red value of the color value **c**:

```
float r1 = red(c);  // Simpler, but slower to calculate
float r2 = c >> 16 & 0xFF;  // Very fast to calculate
```

Syntax `red(rgb)`

Parameters **rgb** int: any value of the color datatype

Returns float

Related green()
blue()
alpha()
hue()
saturation()
brightness()
>> (right shift)

Name

saturation()

Examples

```
noStroke();
colorMode(HSB, 255);
color c = color(0, 126, 255);
fill(c);
rect(15, 20, 35, 60);
float value = saturation(c);  // Sets 'value' to 126
fill(value);
rect(50, 20, 35, 60);
```

Description Extracts the saturation value from a color.

Syntax `saturation(rgb)`

Parameters **rgb** int: any value of the color datatype

Returns float

Related red()
green()
blue()
alpha()
hue()
brightness()

Name

createImage()

Examples

```
PImage img = createImage(66, 66, RGB);
img.loadPixels();
for (int i = 0; i < img.pixels.length; i++) {
  img.pixels[i] = color(0, 90, 102);
}
img.updatePixels();
image(img, 17, 17);
```

```
PImage img = createImage(66, 66, ARGB);
img.loadPixels();
for (int i = 0; i < img.pixels.length; i++) {
  img.pixels[i] = color(0, 90, 102, i % img.width * 2);
}
img.updatePixels();
image(img, 17, 17);
image(img, 34, 34);
```

Description

Creates a new PImage (the datatype for storing images). This provides a fresh buffer of pixels to play with. Set the size of the buffer with the **width** and **height** parameters. The **format** parameter defines how the pixels are stored. See the PImage reference for more information.

Be sure to include all three parameters, specifying only the width and height (but no format) will produce a strange error.

Advanced users please note that createImage() should be used instead of the syntax `new PImage()`.

Syntax

`createImage(w, h, format)`

Parameters

w int: width in pixels
h int: height in pixels
format int: Either RGB, ARGB, ALPHA (grayscale alpha channel)

Returns PImage

Related PImage
PGraphics

Name

PImage

Examples

```
PImage img;

void setup() {
  img = loadImage("laDefense.jpg");
}

void draw() {
  image(img, 0, 0);
}
```

Description

Datatype for storing images. Processing can display **.gif**, **.jpg**, **.tga**, and **.png** images. Images may be displayed in 2D and 3D space. Before an image is used, it must be loaded with the **loadImage()** function. The **PImage** class contains fields for the **width** and **height** of the image, as well as an array called **pixels[]** that contains the values for every pixel in the image. The methods described below allow easy access to the image's pixels and alpha channel and simplify the process of compositing.

Before using the **pixels[]** array, be sure to use the **loadPixels()** method on the image to make sure that the pixel data is properly loaded.

To create a new image, use the **createImage()** function. Do not use the syntax **new PImage()**.

Fields

pixels[]	Array containing the color of every pixel in the image
width	Image width
height	Image height

Methods

loadPixels()	Loads the pixel data for the image into its pixels[] array
updatePixels()	Updates the image with the data in its pixels[] array
resize()	Changes the size of an image to a new width and height
get()	Reads the color of any pixel or grabs a rectangle of pixels
set()	writes a color to any pixel or writes an image into another
mask()	Masks part of an image with another image as an alpha channel
filter()	Converts the image to grayscale or black and white
copy()	Copies the entire image
blend()	Copies a pixel or rectangle of pixels using different blending modes
save()	Saves the image to a TIFF, TARGA, PNG, or JPEG file

Constructor

```
PImage()
PImage(width, height)
PImage(width, height, format)
PImage(img)
```

Parameters

width	int: image width
height	int: image height
format	int: Either RGB, ARGB, ALPHA (grayscale alpha channel)
img	Image: assumes a MediaTracker has been used to fully download the data and the img is valid

Related

loadImage()
imageMode()
createImage()

Name

image()

Examples

```
PImage img;

void setup() {
  // Images must be in the "data" directory to load correctly
  img = loadImage("laDefense.jpg");
}

void draw() {
  image(img, 0, 0);
}
```

```
PImage img;

void setup() {
  // Images must be in the "data" directory to load correctly
```

```
  img = loadImage("laDefense.jpg");
}

void draw() {
  image(img, 0, 0);
  image(img, 0, 0, width/2, height/2);
}
```

The **image()** function draws an image to the display window. Images must be in the sketch's "data" directory to load correctly. Select "Add file..." from the "Sketch" menu to add the image to the data directory, or just drag the image file onto the sketch window. Processing currently works with GIF, JPEG, and PNG images.

Description

The **img** parameter specifies the image to display and by default the **a** and **b** parameters define the location of its upper-left corner. The image is displayed at its original size unless the **c** and **d** parameters specify a different size. The **imageMode()** function can be used to change the way these parameters draw the image.

The color of an image may be modified with the **tint()** function. This function will maintain transparency for GIF and PNG images.

Syntax

```
image(img, a, b)
image(img, a, b, c, d)
```

Parameters

img PImage: the image to display
a float: x-coordinate of the image
b float: y-coordinate of the image
c float: width to display the image
d float: height to display the image

Returns

void

Related

loadImage()
PImage
imageMode()
tint()
background()
alpha()

Name

imageMode()

Examples

```
PImage img;

void setup() {
  img = loadImage("laDefense.jpg");
}

void draw() {
  imageMode(CORNER);
  image(img, 10, 10, 50, 50);  // Draw image using CORNER mode
}


PImage img;
```

```
void setup() {
  img = loadImage("laDefense.jpg");
}

void draw() {
  imageMode(CORNERS);
  image(img, 10, 10, 90, 40);  // Draw image using CORNERS mode
}
```

```
PImage img;

void setup() {
  img = loadImage("laDefense.jpg");
}

void draw() {
  imageMode(CENTER);
  image(img, 50, 50, 80, 80);  // Draw image using CENTER mode
}
```

Description

Modifies the location from which images are drawn by changing the way in which parameters given to **image()** are intepreted.

The default mode is **imageMode(CORNER)**, which interprets the second and third parameters of **image()** as the upper-left corner of the image. If two additional parameters are specified, they are used to set the image's width and height.

imageMode(CORNERS) interprets the second and third parameters of **image()** as the location of one corner, and the fourth and fifth parameters as the opposite corner.

imageMode(CENTER) interprets the second and third parameters of **image()** as the image's center point. If two additional parameters are specified, they are used to set the image's width and height.

The parameter must be written in ALL CAPS because Processing is a case-sensitive language.

Syntax `imageMode(mode)`

Parameters **mode** int: either CORNER, CORNERS, or CENTER

Returns void

Related loadImage()
PImage
image()
background()

Name

loadImage()

Examples

```
PImage img;
img = loadImage("laDefense.jpg");
image(img, 0, 0);
```

```
PImage img;

void setup() {
  img = loadImage("laDefense.jpg");
}

void draw() {
  image(img, 0, 0);
}
```

```
PImage webImg;

void setup() {
  String url = "http://processing.org/img/processing_cover";
  // Load image from a web server
  webImg = loadImage(url, "gif");
}

void draw() {
  background(0);
  image(webImg, 0, 0);
}
```

Loads an image into a variable of type **PImage**. Four types of images (**.gif**, **.jpg**, **.tga**, **.png**) images may be loaded. To load correctly, images must be located in the data directory of the current sketch.

In most cases, load all images in **setup()** to preload them at the start of the program. Loading images inside **draw()** will reduce the speed of a program. Images cannot be loaded outside **setup()** unless they're inside a function that's called after **setup()** has already run.

Alternatively, the file maybe be loaded from anywhere on the local computer using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows), or the filename parameter can be a URL for a file found on a network.

Description If the file is not available or an error occurs, **null** will be returned and an error message will be printed to the console. The error message does not halt the program, however the null value may cause a NullPointerException if your code does not check whether the value returned is null.

The **extension** parameter is used to determine the image type in cases where the image filename does not end with a proper extension. Specify the extension as the second parameter to **loadImage()**, as shown in the third example on this page.

Depending on the type of error, a **PImage** object may still be returned, but the width and height of the image will be set to -1. This happens if bad image data is returned or cannot be decoded properly. Sometimes this happens with image URLs that produce a 403 error or that redirect to a password prompt, because **loadImage()** will attempt to interpret the HTML as image data.

Syntax
```
loadImage(filename)
loadImage(filename, extension)
```

Parameters **filename** String: name of file to load, can be .gif, .jpg, .tga, or a handful of other

image types depending on your platform

extension String: type of image to load, for example "png", "gif", "jpg"

Returns PImage

Related
PImage
image()
imageMode()
background()

Name

noTint()

Examples

```
PImage img;
img = loadImage("laDefense.jpg");
tint(0, 153, 204);  // Tint blue
image(img, 0, 0);
noTint();  // Disable tint
image(img, 50, 0);
```

Description Removes the current fill value for displaying images and reverts to displaying images with their original hues.

Syntax

```
noTint()
```

Returns void

Related
tint()
image()

Name

requestImage()

Examples

```
PImage bigImage;

void setup() {
  bigImage = requestImage("something.jpg");
}

void draw() {
  if (bigImage.width == 0) {
    // Image is not yet loaded
  } else if (bigImage.width == -1) {
    // This means an error occurred during image loading
  } else {
    // Image is ready to go, draw it
    image(bigImage, 0, 0);
  }
}
```

Description This function loads images on a separate thread so that your sketch doesn't freeze while images load during **setup()**. While the image is loading, its width and height will be 0. If an error occurs while loading the image, its width and height will be set to -1. You'll know when the image has loaded properly because its **width** and **height** will be greater than 0. Asynchronous image loading (particularly when downloading from a server) can dramatically improve performance.

The **extension** parameter is used to determine the image type in cases where the image filename does not end with a proper extension. Specify the extension as the second parameter to **requestImage()**.

Syntax

```
requestImage(filename)
requestImage(filename, extension)
```

Parameters

filename	String: name of the file to load, can be .gif, .jpg, .tga, or a handful of other image types depending on your platform
extension	String: the type of image to load, for example "png", "gif", "jpg"

Returns PImage

Related PImage
loadImage()

Name

tint()

Examples

```
PImage img;
img = loadImage("laDefense.jpg");
image(img, 0, 0);
tint(0, 153, 204);  // Tint blue
image(img, 50, 0);
```

```
PImage img;
img = loadImage("laDefense.jpg");
image(img, 0, 0);
tint(0, 153, 204, 126);  // Tint blue and set transparency
image(img, 50, 0);
```

```
PImage img;
img = loadImage("laDefense.jpg");
image(img, 0, 0);
tint(255, 126);  // Apply transparency without changing color
image(img, 50, 0);
```

Description

Sets the fill value for displaying images. Images can be tinted to specified colors or made transparent by including an alpha value.

To apply transparency to an image without affecting its color, use white as the tint color and specify an alpha value. For instance, tint(255, 128) will make an image 50% transparent (assuming the default alpha range of 0-255, which can be changed with **colorMode()**).

When using hexadecimal notation to specify a color, use "#" or "0x" before the values (e.g. #CCFFAA, 0xFFCCFFAA). The # syntax uses six digits to specify a color (the way colors are specified in HTML and CSS). When using the hexadecimal notation starting with "0x", the hexadecimal value must be specified with eight characters; the first two characters define the alpha component and the remainder the red, green, and blue components.

The value for the gray parameter must be less than or equal to the current maximum value as specified by **colorMode()**. The default maximum value is 255.

The **tint()** function is also used to control the coloring of textures in 3D.

Syntax

```
tint(rgb)
tint(rgb, alpha)
tint(gray)
tint(gray, alpha)
tint(v1, v2, v3)
tint(v1, v2, v3, alpha)
```

Parameters

rgb	int: color value in hexadecimal notation
alpha	float: opacity of the image
gray	float: specifies a value between white and black
v1	float: red or hue value (depending on current color mode)
v2	float: green or saturation value (depending on current color mode)
v3	float: blue or brightness value (depending on current color mode)

Returns void

Related
noTint()
image()

Name

texture()

Examples

```
size(100, 100, P3D);
noStroke();
PImage img = loadImage("laDefense.jpg");
beginShape();
texture(img);
vertex(10, 20, 0, 0);
vertex(80, 5, 100, 0);
vertex(95, 90, 100, 100);
vertex(40, 95, 0, 100);
endShape();
```

Description

Sets a texture to be applied to vertex points. The **texture()** function must be called between **beginShape()** and **endShape()** and before any calls to **vertex()**. This function only works with the P2D and P3D renderers.

When textures are in use, the fill color is ignored. Instead, use **tint()** to specify the color of the texture as it is applied to the shape.

Syntax

```
texture(image)
```

Parameters

image	PImage: reference to a PImage object

Returns void

Related
textureMode()
textureWrap()
beginShape()
endShape()
vertex()

Name

textureMode()

```
size(100, 100, P3D);
noStroke();
PImage img = loadImage("laDefense.jpg");
textureMode(IMAGE);
beginShape();
texture(img);
vertex(10, 20, 0, 0);
vertex(80, 5, 100, 0);
vertex(95, 90, 100, 100);
vertex(40, 95, 0, 100);
endShape();
```

Examples

```
size(100, 100, P3D);
noStroke();
PImage img = loadImage("laDefense.jpg");
textureMode(NORMAL);
beginShape();
texture(img);
vertex(10, 20, 0, 0);
vertex(80, 5, 1, 0);
vertex(95, 90, 1, 1);
vertex(40, 95, 0, 1);
endShape();
```

Description Sets the coordinate space for texture mapping. The default mode is **IMAGE**, which refers to the actual coordinates of the image. **NORMAL** refers to a normalized space of values ranging from 0 to 1. This function only works with the P2D and P3D renderers.

With **IMAGE**, if an image is 100 x 200 pixels, mapping the image onto the entire size of a quad would require the points (0,0) (0,100) (100,200) (0,200). The same mapping in **NORMAL** is (0,0) (0,1) (1,1) (0,1).

Syntax `textureMode(mode)`

Parameters **mode** int: either IMAGE or NORMAL

Returns void

Related texture()
textureWrap()

Name

textureWrap()

```
PImage img;

void setup() {
  size(300, 300, P2D);
  img = loadImage("berlin-1.jpg");
  textureMode(NORMAL);
}
```

Examples

```
void draw() {
  background(0);
  translate(width/2, height/2);
  rotate(map(mouseX, 0, width, -PI, PI));
  if (mousePressed) {
    textureWrap(REPEAT);
```

```
  } else {
    textureWrap(CLAMP);
  }
  beginShape();
  texture(img);
  vertex(-100, -100, 0, 0);
  vertex(100, -100, 2, 0);
  vertex(100, 100, 2, 2);
  vertex(-100, 100, 0, 2);
  endShape();
}
```

Description Defines if textures repeat or draw once within a texture map. The two parameters are CLAMP (the default behavior) and REPEAT. This function only works with the P2D and P3D renderers.

Syntax `textureWrap(wrap)`

Parameters **wrap** int: Either CLAMP (default) or REPEAT

Returns void

Related texture()
textureMode()

Name

blend()

Examples

```
background(loadImage("rockies.jpg"));
PImage img = loadImage("bricks.jpg");
image(img, 0, 0);
blend(img, 0, 0, 33, 100, 67, 0, 33, 100, ADD);
```

```
background(loadImage("rockies.jpg"));
PImage img = loadImage("bricks.jpg");
image(img, 0, 0);
blend(img, 0, 0, 33, 100, 67, 0, 33, 100, SUBTRACT);
```

```
background(loadImage("rockies.jpg"));
PImage img = loadImage("bricks.jpg");
image(img, 0, 0);
blend(img, 0, 0, 33, 100, 67, 0, 33, 100, DARKEST);
```

```
background(loadImage("rockies.jpg"));
PImage img = loadImage("bricks.jpg");
image(img, 0, 0);
blend(img, 0, 0, 33, 100, 67, 0, 33, 100, LIGHTEST);
```

Description Blends a region of pixels from one image into another (or in itself again) with full alpha channel support. There is a choice of the following modes to blend the source pixels (A) with the ones of pixels in the destination image (B):

BLEND - linear interpolation of colours: C = A*factor + B

ADD - additive blending with white clip: C = min(A*factor + B, 255)

SUBTRACT - subtractive blending with black clip: C = max(B - A*factor, 0)

DARKEST - only the darkest colour succeeds: C = min(A*factor, B)

LIGHTEST - only the lightest colour succeeds: C = max(A*factor, B)

DIFFERENCE - subtract colors from underlying image.

EXCLUSION - similar to DIFFERENCE, but less extreme.

MULTIPLY - Multiply the colors, result will always be darker.

SCREEN - Opposite multiply, uses inverse values of the colors.

OVERLAY - A mix of MULTIPLY and SCREEN. Multiplies dark values, and screens light values.

HARD_LIGHT - SCREEN when greater than 50% gray, MULTIPLY when lower.

SOFT_LIGHT - Mix of DARKEST and LIGHTEST. Works like OVERLAY, but not as harsh.

DODGE - Lightens light tones and increases contrast, ignores darks. Called "Color Dodge" in Illustrator and Photoshop.

BURN - Darker areas are applied, increasing contrast, ignores lights. Called "Color Burn" in Illustrator and Photoshop.

All modes use the alpha information (highest byte) of source image pixels as the blending factor. If the source and destination regions are different sizes, the image will be automatically resized to match the destination size. If the **srcImg** parameter is not used, the display window is used as the source image.

As of release 0149, this function ignores **imageMode()**.

Syntax

```
blend(sx, sy, sw, sh, dx, dy, dw, dh, mode)
blend(src, sx, sy, sw, sh, dx, dy, dw, dh, mode)
```

Parameters

src	PImage: an image variable referring to the source image
sx	int: X coordinate of the source's upper left corner
sy	int: Y coordinate of the source's upper left corner
sw	int: source image width
sh	int: source image height
dx	int: X coordinate of the destinations's upper left corner
dy	int: Y coordinate of the destinations's upper left corner
dw	int: destination image width
dh	int: destination image height
mode	int: Either BLEND, ADD, SUBTRACT, LIGHTEST, DARKEST, DIFFERENCE, EXCLUSION, MULTIPLY, SCREEN, OVERLAY, HARD_LIGHT, SOFT_LIGHT, DODGE, BURN

Returns void

Related alpha()
copy()

Name

copy()

Examples

```
PImage img = loadImage("eames.jpg");
image(img, 0, 0, width, height);
copy(7, 22, 10, 10, 35, 25, 50, 50);
stroke(255);
noFill();
// Rectangle shows area being copied
rect(7, 22, 10, 10);
```

Description

Copies a region of pixels from the display window to another area of the display window and copies a region of pixels from an image used as the **srcImg** parameter into the display window. If the source and destination regions aren't the same size, it will automatically resize the source pixels to fit the specified target region. No alpha information is used in the process, however if the source image has an alpha channel set, it will be copied as well.

As of release 0149, this function ignores **imageMode()**.

Syntax

```
copy(sx, sy, sw, sh, dx, dy, dw, dh)
copy(src, sx, sy, sw, sh, dx, dy, dw, dh)
```

Parameters

sx int: X coordinate of the source's upper left corner
sy int: Y coordinate of the source's upper left corner
sw int: source image width
sh int: source image height
dx int: X coordinate of the destination's upper left corner
dy int: Y coordinate of the destination's upper left corner
dw int: destination image width
dh int: destination image height
src PImage: an image variable referring to the source image.

Returns

void

Related

alpha()
blend()

Name

filter()

Examples

```
PImage img;
img = loadImage("apples.jpg");
image(img, 0, 0);
filter(THRESHOLD);
```

```
PImage img;
img = loadImage("apples.jpg");
image(img, 0, 0);
filter(GRAY);
```

```
PImage img;
img = loadImage("apples.jpg");
image(img, 0, 0);
filter(INVERT);
```

```
PImage img;
img = loadImage("apples.jpg");
image(img, 0, 0);
filter(POSTERIZE, 4);
```

```
PImage img;
img = loadImage("apples.jpg");
image(img, 0, 0);
filter(BLUR, 6);
```

```
PImage img;
img = loadImage("apples.jpg");
image(img, 0, 0);
filter(ERODE);
```

```
PImage img;
img = loadImage("apples.jpg");
image(img, 0, 0);
filter(DILATE);
```

```
PShader blur;
PImage img;

void setup() {
  size(100, 100, P2D);
  blur = loadShader("blur.glsl");
  img = loadImage("apples.jpg");
  image(img, 0, 0);
}

void draw() {
  filter(blur); // Blurs more each time through draw()
}
```

Filters the display window using a preset filter or with a custom shader. Using a shader with **filter()** is much faster than without. Shaders require the P2D or P3D renderer in size().

The presets options are:

THRESHOLD

Description Converts the image to black and white pixels depending if they are above or below the threshold defined by the level parameter. The parameter must be between 0.0 (black) and 1.0 (white). If no level is specified, 0.5 is used.

GRAY

Converts any colors in the image to grayscale equivalents. No parameter is used.

OPAQUE

Sets the alpha channel to entirely opaque. No parameter is used.

INVERT
Sets each pixel to its inverse value. No parameter is used.

POSTERIZE
Limits each channel of the image to the number of colors specified as the parameter. The parameter can be set to values between 2 and 255, but results are most noticeable in the lower ranges.

BLUR
Executes a Guassian blur with the level parameter specifying the extent of the blurring. If no parameter is used, the blur is equivalent to Guassian blur of radius 1. Larger values increase the blur.

ERODE
Reduces the light areas. No parameter is used.

DILATE
Increases the light areas. No parameter is used.

Syntax

```
filter(shader)
filter(kind)
filter(kind, param)
```

Parameters

shader PShader: the fragment shader to apply
kind int: Either THRESHOLD, GRAY, OPAQUE, INVERT, POSTERIZE, BLUR, ERODE, or DILATE
param float: unique for each, see above

Returns void

Name

get()

Examples

```
PImage myImage = loadImage("apples.jpg");
image(myImage, 0, 0);
PImage c = get();
image(c, width/2, 0);
```

```
PImage myImage = loadImage("apples.jpg");
image(myImage, 0, 0);
color c = get(25, 25);
fill(c);
noStroke();
rect(25, 25, 50, 50);
```

Description

Reads the color of any pixel or grabs a section of an image. If no parameters are specified, the entire image is returned. Use the **x** and **y** parameters to get the value of one pixel. Get a section of the display window by specifying additional **w** and **h** parameters. When getting an image, the **x** and **y** parameters define the coordinates for the upper-left corner of the image, regardless of the current **imageMode()**.

If the pixel requested is outside of the image window, black is returned. The numbers returned are scaled according to the current color ranges, but only RGB values are returned by this function. For example, even though you may have drawn a shape with **colorMode(HSB)**, the numbers returned will be in RGB format.

Getting the color of a single pixel with **get(x, y)** is easy, but not as fast as grabbing the data directly from **pixels[]**. The equivalent statement to **get(x, y)** using **pixels[]** is **pixels[y*width+x]**. See the reference for pixels[] for more information.

Syntax

```
get(x, y)
get(x, y, w, h)
get()
```

Parameters

x int: x-coordinate of the pixel
y int: y-coordinate of the pixel
w int: width of pixel rectangle to get
h int: height of pixel rectangle to get

Returns int or PImage

Related set()
pixels[]
copy()

Name

loadPixels()

Examples

```
int halfImage = width*height/2;
PImage myImage = loadImage("apples.jpg");
image(myImage, 0, 0);

loadPixels();
for (int i = 0; i < halfImage; i++) {
  pixels[i+halfImage] = pixels[i];
}
updatePixels();
```

Description Loads the pixel data for the display window into the **pixels[]** array. This function must always be called before reading from or writing to **pixels[]**.

Certain renderers may or may not seem to require **loadPixels()** or **updatePixels()**. However, the rule is that any time you want to manipulate the **pixels[]** array, you must first call **loadPixels()**, and after changes have been made, call **updatePixels()**. Even if the renderer may not seem to use this function in the current Processing release, this will always be subject to change.

Syntax

```
loadPixels()
```

Returns void

Related pixels[]
updatePixels()

Name

pixels[]

Examples

```
color pink = color(255, 102, 204);
loadPixels();
for (int i = 0; i < (width*height/2)-width/2; i++) {
  pixels[i] = pink;
}
updatePixels();
```

Description

Array containing the values for all the pixels in the display window. These values are of the color datatype. This array is the size of the display window. For example, if the image is 100x100 pixels, there will be 10000 values and if the window is 200x300 pixels, there will be 60000 values. The **index** value defines the position of a value within the array. For example, the statement **color b = pixels[230]** will set the variable **b** to be equal to the value at that location in the array.

Before accessing this array, the data must loaded with the **loadPixels()** function. After the array data has been modified, the **updatePixels()** function must be run to update the changes. Without **loadPixels()**, running the code may (or will in future releases) result in a NullPointerException.

Related

loadPixels()
updatePixels()
get()
set()
PImage

Name

set()

Examples

```
color black = color(0);
set(30, 20, black);
set(85, 20, black);
set(85, 75, black);
set(30, 75, black);
```

```
for (int i = 30; i < width-15; i++) {
  for (int j = 20; j < height-25; j++) {
    color c = color(204-j, 153-i, 0);
    set(i, j, c);
  }
}
```

```
size(100, 100);
PImage myImage = loadImage("apples.jpg");
set(0, 0, myImage);
line(0, 0, width, height);
line(0, height, width, 0);
```

Description

Changes the color of any pixel, or writes an image directly to the display window.

The **x** and **y** parameters specify the pixel to change and the **c** parameter specifies the color value. The **c** parameter is interpreted according to the current color

mode. (The default color mode is RGB values from 0 to 255.) When setting an image, the **x** and **y** parameters define the coordinates for the upper-left corner of the image, regardless of the current **imageMode()**.

Setting the color of a single pixel with **set(x, y)** is easy, but not as fast as putting the data directly into **pixels[]**. The equivalent statement to **set(x, y, #000000)** using **pixels[]** is **pixels[y*width+x] = #000000**. See the reference for pixels[] for more information.

Syntax

```
set(x, y, c)
set(x, y, img)
```

Parameters

x int: x-coordinate of the pixel
y int: y-coordinate of the pixel
c int: any value of the color datatype
img PImage: image to copy into the original image

Returns void

Related get()
pixels[]
copy()

Name

updatePixels()

Examples

```
PImage img = loadImage("rockies.jpg");
image(img, 0, 0);
int halfImage = img.width * img.height/2;
loadPixels();
for (int i = 0; i < halfImage; i++) {
  pixels[i+halfImage] = pixels[i];
}
updatePixels();
```

Description Updates the display window with the data in the **pixels[]** array. Use in conjunction with **loadPixels()**. If you're only reading pixels from the array, there's no need to call **updatePixels()** — updating is only necessary to apply changes.

Certain renderers may or may not seem to require **loadPixels()** or **updatePixels()**. However, the rule is that any time you want to manipulate the **pixels[]** array, you must first call **loadPixels()**, and after changes have been made, call **updatePixels()**. Even if the renderer may not seem to use this function in the current Processing release, this will always be subject to change.

Currently, while none of the renderers use the additional parameters to **updatePixels()**, this may be implemented in the future.

Syntax

```
updatePixels()
```

Returns void

Related loadPixels()
pixels[]

Name

blendMode()

Examples

```
size(100, 100);
background(0);
blendMode(ADD);
stroke(102);
strokeWeight(30);
line(25, 25, 75, 75);
line(75, 25, 25, 75);
```

```
size(100, 100, P2D);
blendMode(MULTIPLY);
stroke(51);
strokeWeight(30);
line(25, 25, 75, 75);
line(75, 25, 25, 75);
```

Description Blends the pixels in the display window according to the defined mode. There is a choice of the following modes to blend the source pixels (A) with the ones of pixels already in the display window (B):

BLEND - linear interpolation of colours: C = A*factor + B. This is the default blending mode.

ADD - additive blending with white clip: C = min(A*factor + B, 255)

SUBTRACT - subtractive blending with black clip: C = max(B - A*factor, 0)

DARKEST - only the darkest colour succeeds: C = min(A*factor, B)

LIGHTEST - only the lightest colour succeeds: C = max(A*factor, B)

DIFFERENCE - subtract colors from underlying image.

EXCLUSION - similar to DIFFERENCE, but less extreme.

MULTIPLY - multiply the colors, result will always be darker.

SCREEN - opposite multiply, uses inverse values of the colors.

REPLACE - the pixels entirely replace the others and don't utilize alpha (transparency) values

For Processing 2.0, we recommend using **blendMode()** and not the previous **blend()** function. However, unlike **blend()**, the **blendMode()** function does not support the following: HARD_LIGHT, SOFT_LIGHT, OVERLAY, DODGE, BURN. On older hardware, the LIGHTEST, DARKEST, and DIFFERENCE modes might not be available as well.

Syntax `blendMode(mode)`

Parameters **mode** int: the blending mode to use

Returns void

Name

createGraphics()

```
PGraphics pg;

void setup() {
  size(200, 200);
  pg = createGraphics(100, 100);
}
```

Examples

```
void draw() {
  pg.beginDraw();
  pg.background(102);
  pg.stroke(255);
  pg.line(pg.width*0.5, pg.height*0.5, mouseX, mouseY);
  pg.endDraw();
  image(pg, 50, 50);
}
```

Description

Creates and returns a new **PGraphics** object. Use this class if you need to draw into an off-screen graphics buffer. The first two parameters define the width and height in pixels. The third, optional parameter specifies the renderer. It can be defined as P2D, P3D, or PDF. If the third parameter isn't used, the default renderer is set. The PDF renderer requires the filename parameter.

It's important to consider the renderer used with **createGraphics()** in relation to the main renderer specified in **size()**. For example, it's only possible to use P2D or P3D with **createGraphics()** when one of them is defined in **size()**. Unlike Processing 1.0, P2D and P3D use OpenGL for drawing, and when using an OpenGL renderer it's necessary for the main drawing surface to be OpenGL-based. If P2D or P3D are used as the renderer in **size()**, then any of the options can be used with **createGraphics()**. If the default renderer is used in **size()**, then only the default or PDF can be used with **createGraphics()**.

It's important to call any drawing functions between **beginDraw()** and **endDraw()** statements. This is also true for any functions that affect drawing, such as **smooth()** or **colorMode()**.

Unlike the main drawing surface which is completely opaque, surfaces created with **createGraphics()** can have transparency. This makes it possible to draw into a graphics and maintain the alpha channel. By using **save()** to write a PNG or TGA file, the transparency of the graphics object will be honored.

Syntax

```
createGraphics(w, h)
createGraphics(w, h, renderer)
createGraphics(w, h, renderer, path)
```

Parameters

- **w** int: width in pixels
- **h** int: height in pixels
- **renderer** String: Either P2D, P3D, or PDF
- **path** String: the name of the file (can be an absolute or relative path)

Returns PGraphics

Name

PGraphics

```
PGraphics pg;
```

Examples

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

```
  pg = createGraphics(40, 40);
}

void draw() {
  pg.beginDraw();
  pg.background(100);
  pg.stroke(255);
  pg.line(20, 20, mouseX, mouseY);
  pg.endDraw();
  image(pg, 9, 30);
  image(pg, 51, 30);
}
```

Description Main graphics and rendering context, as well as the base API implementation for processing "core". Use this class if you need to draw into an off-screen graphics buffer. A PGraphics object can be constructed with the **createGraphics()** function. The **beginDraw()** and **endDraw()** methods (see above example) are necessary to set up the buffer and to finalize it. The fields and methods for this class are extensive. For a complete list, visit the developer's reference.

Methods **beginDraw()** Sets the default properties for a PGraphics object
endDraw() Finalizes the rendering of a PGraphics object

Constructor `PGraphics()`

Related createGraphics()

Name

loadShader()

```
PShader blur;

void setup() {
  size(640, 360, P2D);
  // Shaders files must be in the "data" folder to load correctly
  blur = loadShader("blur.glsl");
  stroke(0, 102, 153);
  rectMode(CENTER);
}

void draw() {
  filter(blur);
  rect(mouseX-75, mouseY, 150, 150);
  ellipse(mouseX+75, mouseY, 150, 150);
}
```

Examples

Description Loads a shader into the PShader object. The shader file must be loaded in the sketch's "data" folder/directory to load correctly. Shaders are compatible with the P2D and P3D renderers, but not with the default renderer.

Alternatively, the file maybe be loaded from anywhere on the local computer using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows), or the filename parameter can be a URL for a file found on a network.

If the file is not available or an error occurs, **null** will be returned and an error message will be printed to the console. The error message does not halt the program, however the null value may cause a NullPointerException if your code does not check whether the value returned is null.

Syntax `loadShader(fragFilename)`

```
loadShader(fragFilename, vertFilename)
```

Parameters
fragFilename String: name of fragment shader file
vertFilename String: name of vertex shader file

Returns PShader

Name

PShader

Examples

```
PShader blur;

void setup() {
  size(640, 360, P2D);
  // Shaders files must be in the "data" folder to load correctly
  blur = loadShader("blur.glsl");
  stroke(0, 102, 153);
  rectMode(CENTER);
}

void draw() {
  filter(blur);
  rect(mouseX-75, mouseY, 150, 150);
  ellipse(mouseX+75, mouseY, 150, 150);
}
```

Description This class encapsulates a GLSL shader program, including a vertex and a fragment shader. It's compatible with the P2D and P3D renderers, but not with the default renderer. Use the **loadShader()** function to load your shader code. [Note: It's strongly encouraged to use **loadShader()** to create a PShader object, rather than calling the PShader constructor manually.]

Methods set() Sets a variable within the shader

Constructor

```
PShader()
PShader(parent)
PShader(parent, vertFilename, fragFilename)
PShader(parent, vertURL, fragURL)
```

Parameters
parent PApplet: the parent program
vertFilename String: name of the vertex shader
fragFilename String: name of the fragment shader
vertURL URL: network location of the vertex shader
fragURL URL: network location of the fragment shader

Name

resetShader()

Examples

```
PShader edges;
PImage img;

void setup() {
  size(640, 360, P2D);
  img = loadImage("leaves.jpg");
  edges = loadShader("edges.glsl");
}

void draw() {
  shader(edges);
  image(img, 0, 0);
  resetShader();
  image(img, width/2, 0);
```

```
}
```

Description Restores the default shaders. Code that runs after **resetShader()** will not be affected by previously defined shaders.

Syntax
```
resetShader()
resetShader(kind)
```

Parameters **kind** int: type of shader, either POINTS, LINES, or TRIANGLES

Returns void

Name

shader()

Examples
```
PShader edges;
PImage img;

void setup() {
  size(640, 360, P2D);
  img = loadImage("leaves.jpg");
  edges = loadShader("edges.glsl");
}

void draw() {
  shader(edges);
  image(img, 0, 0);
}
```

Description Applies the shader specified by the parameters. It's compatible with the P2D and P3D renderers, but not with the default renderer.

Syntax
```
shader(shader)
shader(shader, kind)
```

Parameters **shader** PShader: name of shader file
kind int: type of shader, either POINTS, LINES, or TRIANGLES

Returns void

Name

PFont

Examples
```
PFont font;
// The font must be located in the sketch's
// "data" directory to load successfully
font = loadFont("LetterGothicStd-32.vlw");
textFont(font, 32);
text("word", 10, 50);
```

Description PFont is the font class for Processing. To create a font to use with Processing, select "Create Font..." from the Tools menu. This will create a font in the format Processing requires and also adds it to the current sketch's data directory. Processing displays fonts using the .vlw font format, which uses images for each letter, rather than defining them through vector data. The **loadFont()** function constructs a new font and **textFont()** makes a font active. The **list()** method creates a list of the fonts installed on the computer, which is useful information to use with the **createFont()** function for dynamically converting fonts into a format to use with Processing.

Methods list() Gets a list of the fonts installed on the system

Constructor
```
PFont()
PFont(font, smooth)
```

```
PFont(font, smooth, charset)
PFont(font, smooth, charset, stream)
PFont(input)
```

Parameters
font Font: font the font object to create from
smooth boolean: smooth true to enable smoothing/anti-aliasing
charset char[]: array of all unicode chars that should be included
input InputStream: InputStream

Related loadFont()

Name

createFont()

Examples

```
PFont myFont;

void setup() {
  size(200, 200);
  // Uncomment the following two lines to see the available fonts
  //String[] fontList = PFont.list();
  //println(fontList);
  myFont = createFont("Georgia", 32);
  textFont(myFont);
  textAlign(CENTER, CENTER);
  text("!@#$%", width/2, height/2);
}
```

Description

Dynamically converts a font to the format used by Processing from a .ttf or .otf file inside the sketch's "data" folder or a font that's installed elsewhere on the computer. If you want to use a font installed on your computer, use the **PFont.list()** method to first determine the names for the fonts recognized by the computer and are compatible with this function. Not all fonts can be used and some might work with one operating system and not others. When sharing a sketch with other people or posting it on the web, you may need to include a .ttf or .otf version of your font in the data directory of the sketch because other people might not have the font installed on their computer. Only fonts that can legally be distributed should be included with a sketch.

The **size** parameter states the font size you want to generate. The **smooth** parameter specifies if the font should be antialiased or not. The **charset** parameter is an array of chars that specifies the characters to generate.

This function allows Processing to work with the font natively in the default renderer, so the letters are defined by vector geometry and are rendered quickly. In the **P2D** and **P3D** renderers, the function sets the project to render the font as a series of small textures. For instance, when using the default renderer, the actual native version of the font will be employed by the sketch, improving drawing quality and performance. With the **P2D** and **P3D** renderers, the bitmapped version will be used to improve speed and appearance, but the results are poor when exporting if the sketch does not include the .otf or .ttf file, and the requested font is not available on the machine running the sketch.

Syntax

```
createFont(name, size)
createFont(name, size, smooth)
createFont(name, size, smooth, charset)
```

Parameters **name** String: name of the font to load

size float: point size of the font
smooth boolean: true for an antialiased font, false for aliased
charset char[]: array containing characters to be generated

Returns PFont

Related
PFont
textFont()
PGraphics
loadFont()

Name

loadFont()

Examples

```
PFont font;
// The font must be located in the sketch's
// "data" directory to load successfully
font = loadFont("LetterGothicStd-32.vlw");
textFont(font, 32);
text("word", 10, 50);
```

Description

Loads a .vlw formatted font into a **PFont** object. Create a .vlw font by selecting "Create Font..." from the Tools menu. This tool creates a texture for each alphanumeric character and then adds them as a .vlw file to the current sketch's data folder. Because the letters are defined as textures (and not vector data) the size at which the fonts are created must be considered in relation to the size at which they are drawn. For example, load a 32pt font if the sketch displays the font at 32 pixels or smaller. Conversely, if a 12pt font is loaded and displayed at 48pts, the letters will be distorted because the program will be stretching a small graphic to a large size.

Like **loadImage()** and other functions that load data, the **loadFont()** function should not be used inside **draw()**, because it will slow down the sketch considerably, as the font will be re-loaded from the disk (or network) on each frame. It's recommended to load files inside **setup()**

To load correctly, fonts must be located in the "data" folder of the current sketch. Alternatively, the file maybe be loaded from anywhere on the local computer using an absolute path (something that starts with / on Unix and Linux, or a drive letter on Windows), or the filename parameter can be a URL for a file found on a network.

If the file is not available or an error occurs, **null** will be returned and an error message will be printed to the console. The error message does not halt the program, however the null value may cause a NullPointerException if your code does not check whether the value returned is null.

Use **createFont()** (instead of **loadFont()**) to enable vector data to be used with the default renderer setting. This can be helpful when many font sizes are needed, or when using any renderer based on the default renderer, such as the PDF library.

Syntax

```
loadFont(filename)
```

Parameters **filename** String: name of the font to load

Returns PFont

Related PFont
textFont()
createFont()

Name

text()

```
textSize(32);
text("word", 10, 30);
fill(0, 102, 153);
text("word", 10, 60);
fill(0, 102, 153, 51);
text("word", 10, 90);
```

Examples

```
size(100, 100, P3D);
textSize(32);
fill(0, 102, 153, 204);
text("word", 12, 45, -30);  // Specify a z-axis value
text("word", 12, 60);  // Default depth, no z-value specified
```

```
String s = "The quick brown fox jumped over the lazy dog.";
fill(50);
text(s, 10, 10, 70, 80);  // Text wraps within text box
```

Draws text to the screen. Displays the information specified in the first parameter on the screen in the position specified by the additional parameters. A default font will be used unless a font is set with the **textFont()** function and a default size will be used unless a font is set with **textSize()**. Change the color of the text with the **fill()** function. The text displays in relation to the **textAlign()** function, which gives the option to draw to the left, right, and center of the coordinates.

Description The **x2** and **y2** parameters define a rectangular area to display within and may only be used with string data. When these parameters are specified, they are interpreted based on the current **rectMode()** setting. Text that does not fit completely within the rectangle specified will not be drawn to the screen.

Note that Processing now lets you call **text()** without first specifying a PFont with **textFont()**. In that case, a generic sans-serif font will be used instead. (See the third example above.)

Syntax

```
text(c, x, y)
text(c, x, y, z)
text(str, x, y)
text(chars, start, stop, x, y)
text(str, x, y, z)
text(chars, start, stop, x, y, z)
text(str, x1, y1, x2, y2)
text(num, x, y)
text(num, x, y, z)
```

Parameters

c char: the alphanumeric character to be displayed
x float: x-coordinate of text

y float: y-coordinate of text
z float: z-coordinate of text
chars char[]: the alphanumberic symbols to be displayed
start int: array index at which to start writing characters
stop int: array index at which to stop writing characters
x1 float: by default, the x-coordinate of text, see rectMode() for more info
y1 float: by default, the x-coordinate of text, see rectMode() for more info
x2 float: by default, the width of the text box, see rectMode() for more info
y2 float: by default, the height of the text box, see rectMode() for more info
num int, or float: the numeric value to be displayed

Returns void

Related
textAlign()
textMode()
loadFont()
textFont()
rectMode()
fill()
String

Name

textFont()

Examples

```
PFont mono;
// The font "AndaleMono-48.vlw"" must be located in the
// current sketch's "data" directory to load successfully
mono = loadFont("AndaleMono-32.vlw");
background(0);
textFont(mono);
text("word", 12, 60);
```

Description

Sets the current font that will be drawn with the **text()** function. Fonts must be created for Processing with **createFont()** or loaded with **loadFont()** before they can be used. The font set through **textFont()** will be used in all subsequent calls to the **text()** function. If no **size** parameter is input, the font will appear at its original size (the size in which it was created with the "Create Font..." tool) until it is changed with **textSize()**.

Because fonts are usually bitmapped, you should create fonts at the sizes that will be used most commonly. Using **textFont()** without the size parameter will result in the cleanest type.

With the default and PDF renderers, it's also possible to enable the use of native fonts via the command **hint(ENABLE_NATIVE_FONTS)**. This will produce vector text in both on-screen sketches and PDF output when the vector data is available, such as when the font is still installed, or the font is created dynamically via the **createFont()** function (rather than with the "Create Font..." tool).

Syntax

```
textFont(which)
textFont(which, size)
```

Parameters

which PFont: any variable of the type PFont
size float: the size of the letters in units of pixels

Returns void

Related createFont()
loadFont()
PFont
text()

Name

textAlign()

Examples

```
PFont font;
// The font must be located in the current sketch's
// "data" directory to load successfully
font = loadFont("LetterGothicStd-32.vlw");
textFont(font, 20);
textAlign(RIGHT);
text("WORD", 50, 30);
textAlign(CENTER);
text("WORD", 50, 50);
textAlign(LEFT);
text("WORD", 50, 70);
```

```
background(0);
stroke(153);
textSize(16);
textAlign(CENTER, BOTTOM);
line(0, 30, width, 30);
text("WORD", 50, 30);
textAlign(CENTER, CENTER);
line(0, 50, width, 50);
text("WORD", 50, 50);
textAlign(CENTER, TOP);
line(0, 70, width, 70);
text("WORD", 50, 70);
```

Description Sets the current alignment for drawing text. The parameters LEFT, CENTER, and RIGHT set the display characteristics of the letters in relation to the values for the **x** and **y** parameters of the **text()** function.

An optional second parameter can be used to vertically align the text. BASELINE is the default, and the vertical alignment will be reset to BASELINE if the second parameter is not used. The TOP and CENTER parameters are straightforward. The BOTTOM parameter offsets the line based on the current **textDescent()**. For multiple lines, the final line will be aligned to the bottom, with the previous lines appearing above it.

When using **text()** with width and height parameters, BASELINE is ignored, and treated as TOP. (Otherwise, text would by default draw outside the box, since BASELINE is the default setting. BASELINE is not a useful drawing mode for text drawn in a rectangle.)

The vertical alignment is based on the value of **textAscent()**, which many fonts do not specify correctly. It may be necessary to use a hack and offset by a few pixels by hand so that the offset looks correct. To do this as less of a hack, use

some percentage of **textAscent()** or **textDescent()** so that the hack works even if you change the size of the font.

Syntax
```
textAlign(alignX)
textAlign(alignX, alignY)
```

Parameters
alignX int: horizontal alignment, either LEFT, CENTER, or RIGHT
alignY int: vertical alignment, either TOP, BOTTOM, CENTER, or BASELINE

Returns void

Related
loadFont()
PFont
text()

Name

textLeading()

Examples
```
// Text to display. The "\n" is a "new line" character
String lines = "L1\nL2\nL3";
textSize(12);
fill(0);  // Set fill to black

textLeading(10);  // Set leading to 10
text(lines, 10, 25);

textLeading(20);  // Set leading to 20
text(lines, 40, 25);

textLeading(30);  // Set leading to 30
text(lines, 70, 25);
```

Description Sets the spacing between lines of text in units of pixels. This setting will be used in all subsequent calls to the **text()** function. Note, however, that the leading is reset by **textSize()**. For example, if the leading is set to 20 with **textLeading(20)**, then if **textSize(48)** is run at a later point, the leading will be reset to the default for the text size of 48.

Syntax
```
textLeading(leading)
```

Parameters **leading** float: the size in pixels for spacing between lines

Returns void

Related
loadFont()
text()
textFont()

Name

textMode()

Examples
```
import processing.pdf.*;

void setup() {
  size(500, 500, PDF, "TypeDemo.pdf");
  textMode(SHAPE);
  textSize(180);
}

void draw() {
  text("ABC", 75, 350);
```

```
  exit();  // Quit the program
}
```

Description Sets the way text draws to the screen, either as texture maps or as vector geometry. The default **textMode(MODEL)**, uses textures to render the fonts. The **textMode(SHAPE)** mode draws text using the glyph outlines of individual characters rather than as textures. This mode is only supported with the **PDF** and **P3D** renderer settings. With the **PDF** renderer, you must call **textMode(SHAPE)** before any other drawing occurs. If the outlines are not available, then **textMode(SHAPE)** will be ignored and **textMode(MODEL)** will be used instead.

The **textMode(SHAPE)** option in **P3D** can be combined with **beginRaw()** to write vector-accurate text to 2D and 3D output files, for instance **DXF** or **PDF**. The **SHAPE** mode is not currently optimized for **P3D**, so if recording shape data, use **textMode(MODEL)** until you're ready to capture the geometry with **beginRaw()**.

Syntax `textMode(mode)`

Parameters **mode** int: either MODEL or SHAPE

Returns void

Related loadFont()
text()
textFont()
beginRaw()
createFont()

Name

textSize()

Examples

```
background(0);
fill(255);
textSize(26);
text("WORD", 10, 50);
textSize(14);
text("WORD", 10, 70);
```

Description Sets the current font size. This size will be used in all subsequent calls to the **text()** function. Font size is measured in units of pixels.

Syntax `textSize(size)`

Parameters **size** float: the size of the letters in units of pixels

Returns void

Related loadFont()
text()
textFont()

Name

textWidth()

Examples

```
textSize(28);
```

```
char c = 'T';
float cw = textWidth(c);
text(c, 0, 40);
line(cw, 0, cw, 50);

String s = "Tokyo";
float sw = textWidth(s);
text(s, 0, 85);
line(sw, 50, sw, 100);
```

Description Calculates and returns the width of any character or text string.

Syntax
```
textWidth(c)
textWidth(str)
```

Parameters **c** char: the character to measure
str String: the String of characters to measure

Returns float

Related loadFont()
text()
textFont()

Name

textAscent()

Examples
```
float base = height * 0.75;
float scalar = 0.8; // Different for each font

textSize(32);  // Set initial text size
float a = textAscent() * scalar;  // Calc ascent
line(0, base-a, width, base-a);
text("dp", 0, base);  // Draw text on baseline

textSize(64);  // Increase text size
a = textAscent() * scalar;  // Recalc ascent
line(40, base-a, width, base-a);
text("dp", 40, base);  // Draw text on baseline
```

Description Returns ascent of the current font at its current size. This information is useful for determining the height of the font above the baseline. For example, adding the **textAscent()** and **textDescent()** values will give you the total height of the line.

Syntax
```
textAscent()
```

Returns float

Related textDescent()

Name

textDescent()

Examples
```
float base = height * 0.75;
float scalar = 0.8; // Different for each font

textSize(32);  // Set initial text size
float a = textDescent() * scalar;  // Calc ascent
line(0, base+a, width, base+a);
text("dp", 0, base);  // Draw text on baseline
```

```
textSize(64);  // Increase text size
a = textDescent() * scalar;  // Recalc ascent
line(40, base+a, width, base+a);
text("dp", 40, base);  // Draw text on baseline
```

Description Returns descent of the current font at its current size. This information is useful for determining the height of the font below the baseline. For example, adding the **textAscent()** and **textDescent()** values will give you the total height of the line.

Syntax `textDescent()`

Returns float

Related textAscent()

Name

PVector

Examples

```
PVector v1, v2;

void setup() {
  noLoop();
  v1 = new PVector(40, 20);
  v2 = new PVector(25, 50);
}

void draw() {
  ellipse(v1.x, v1.y, 12, 12);
  ellipse(v2.x, v2.y, 12, 12);
  v2.add(v1);
  ellipse(v2.x, v2.y, 24, 24);
}
```

A class to describe a two or three dimensional vector, specifically a Euclidean (also known as geometric) vector. A vector is an entity that has both magnitude and direction. The datatype, however, stores the components of the vector (x,y for 2D, and x,y,z for 3D). The magnitude and direction can be accessed via the methods **mag()** and **heading()**.

Description In many of the Processing examples, you will see **PVector** used to describe a position, velocity, or acceleration. For example, if you consider a rectangle moving across the screen, at any given instant it has a position (a vector that points from the origin to its location), a velocity (the rate at which the object's position changes per time unit, expressed as a vector), and acceleration (the rate at which the object's velocity changes per time unit, expressed as a vector). Since vectors represent groupings of values, we cannot simply use traditional addition/multiplication/etc. Instead, we'll need to do some "vector" math, which is made easy by the methods inside the **PVector** class.

Fields

x	The x component of the vector
y	The y component of the vector
z	The z component of the vector

Methods

set()	Set the x, y, and z component of the vector
set()	Set the x, y components of the vector
random2D()	Make a new 2D unit vector with a random direction.
random3D()	Make a new 3D unit vector with a random direction.
fromAngle()	Make a new 2D unit vector from an angle
get()	Get a copy of the vector
mag()	Calculate the magnitude of the vector

magSq()	Calculate the magnitude of the vector, squared
add()	Adds x, y, and z components to a vector, one vector to another, or two independent vectors
sub()	Subtract x, y, and z components from a vector, one vector from another, or two independent vectors
mult()	Multiply a vector by a scalar
div()	Divide a vector by a scalar
dist()	Calculate the distance between two points
dot()	Calculate the dot product of two vectors
cross()	Calculate and return the cross product
normalize()	Normalize the vector to a length of 1
limit()	Limit the magnitude of the vector
setMag()	Set the magnitude of the vector
heading()	Calculate the angle of rotation for this vector
rotate()	Rotate the vector by an angle (2D only)
lerp()	Linear interpolate the vector to another vector
angleBetween()	Calculate and return the angle between two vectors
array()	Return a representation of the vector as a float array

Constructor

```
PVector()
PVector(x, y, z)
PVector(x, y)
```

Parameters

x float: the x coordinate.
y float: the y coordinate.
z float: the z coordinate.

Name

% (modulo)

```
int a = 5 % 4;             // Sets 'a' to 1
int b = 125 % 100;         // Sets 'b' to 25
float c = 285.5 % 140.0;   // Sets 'c' to 5.5
float d = 30.0 % 33.0;     // Sets 'd' to 30.0
```

Examples

```
int a = 0;
void draw() {
  background(200);
  a = (a + 1) % width;  // 'a' increases between 0 and width
  line(a, 0, a, height);
}
```

Description

Calculates the remainder when one number is divided by another. For example, when 52.1 is divided by 10, the divisor (10) goes into the dividend (52.1) five times (5 * 10 == 50), and there is a remainder of 2.1 (52.1 - 50 == 2.1). Thus, **52.1 % 10** produces **2.1**.

Note that when the divisor is greater than the dividend, the remainder constitutes the value of the entire dividend. That is, a number cannot be divided by any number larger than itself. For example, when 9 is divided by 10, the result is zero with a remainder of 9. Thus, **9 % 10** produces **9**.

Modulo is extremely useful for keeping numbers within a boundary such as keeping a shape on the screen. (See the second example above.)

Syntax

```
value1 % value2
```

Parameters **value1** int or float
value2 int or float

Related / (divide)

Name

* (multiply)

Examples
```
int e = 50 * 5;  // Sets 'e' to 250
int f = e * 5;   // Sets 'f' to 1250
```

Description Multiplies the values of the two parameters. Multiplication is equivalent to a sequence of addition. For example 5 * 4 is equivalent to 5 + 5 + 5 + 5.

Syntax
```
value1 * value2
```

Parameters **value1** int, float, byte, or char
value2 int, float, byte, or char

Related + (add)
/ (divide)

Name

*= (multiply assign)

Examples
```
int a = 5;
int b = 2;
a *= b;  // Sets 'a' to 10
```

Description Combines multiplication with assignment. The expression **a *= b** is equivalent to **a = a * b**.

Syntax
```
value1 *= value2
```

Parameters **value1** int or float
value2 any numerical value the same datatype as value1

Related = (assign)
* (multiply)

Name

+ (addition)

Examples
```
int a = 50 + 5;  // Sets 'a' to 55
int b = a + 5;   // Sets 'b' to 60
```

```
String s1 = "Chernenko";
String s2 = "Brezhnev";
String sc1 = s1 + s2;
String sc2 = s1 + ", Andropov, " + s2;
println(sc1);  // Prints "ChernenkoBrezhnev"
println(sc2);  // Prints "Chernenko, Andropov, Brezhnev"
```

```
String s1 = "Gorbachev";
int i = 1987;
String sc1 = s1 + i;
println(sc1);  // Prints "Gorbachev1987"
```

Description Adds two values or concatenates string values. As a mathematical operator, it calculates the sum of two values. As a string operator, it combines two strings into one and converts from primitive datatypes into the String datatype if

necessary.

Syntax
```
value1 + value2
```

Parameters
value1 String, int, float, char, byte, boolean
value2 String, int, float, char, byte, boolean

Related
++ (increment)
+= (add assign)
- (minus)

Name

++ (increment)

Examples
```
int a = 1;   // Sets 'a' to 1
int b = a++; // Sets 'b' to 1, then increments 'a' to 2
int c = a;   // Sets 'c' to 2
```

Description Increases the value of an integer variable by 1. Equivalent to the operation **i = i + 1**. If the value of the variable **i** is five, then the expression **i++** increases the value of **i** to 6.

Syntax
```
value++
```

Parameters
value int

Related
+ (add)
+= (add assign)
-- (decrement)

Name

+= (add assign)

Examples
```
int a = 50;
int b = 23;
a += b;  // Sets 'a' to 73
```

Description Combines addition with assignment. The expression **a += b** is equivalent to **a = a + b**.

Syntax
```
value1 += value2
```

Parameters
value1 int or float
value2 any numerical value the same datatype as value1

Related
= (assign)
+ (add)
-= (subtract assign)

Name

- (minus)

Examples
```
int c = 50 - 5;  // Sets 'c' to 45
int d = c - 5;   // Sets 'd' to 40
int e = d - 60;  // Sets 'e' to -20
```

```
int a = 5;          // Sets 'a' to 5
int b = -a;         // Sets 'b' to -5
int c = -(5 + 3);   // Sets 'c' to -8
```

Description Subtracts one value from another and may also be used to negate a value. As a subtraction operator, the value of the second parameter is subtracted from the

first. For example, 5 - 3 yields the number 2. As a negation operator, it is equivalent to multiplying a number by -1. For example, -5 is the same as 5 * -1.

Syntax
```
-value1
value1 - value2
```

Parameters
value1 int or float
value2 int or float

Related
-- (decrement)
-= (subtract assign)
+ (add)

Name

-- (decrement)

Examples
```
int a = 5;    // Sets 'a' to 5
int b = a--;  // Sets 'b' to 5, then decrements 'a' to 4
int c = a;    // Sets 'c' to 4
```

Description
Substracts the value of an integer variable by 1. Equivalent to the operation **i = i - 1**. If the value of the variable **i** is five, then the expression **i--** decreases the value of **i** to 4.

Syntax
```
var--
```

Parameters
var int

Related
- (minus)
-= (subtract assign)
++ (increment)

Name

-= (subtract assign)

Examples
```
int a = 50;
int b = 23;
a -= b;  // Sets 'a' to 27
```

Description
Combines subtraction with assignment. The expression **a -= b** is equivalent to **a = a - b**.

Syntax
```
value1 -= value2
```

Parameters
value1 int or float
value2 int or float

Related
+= (add assign)
- (minus)

Name

/ (divide)

Examples
```
int g = 50 / 5;  // Assigns 10 to 'g'
int h = g / 5;   // Assigns 2 to 'h'
```

Description
Divides the value of the first parameter by the value of the second parameter. The answer to the equation 20 / 4 is 5. The number 20 is the sum of four occurences of the number 5. As an equation we see that 5 + 5 + 5 + 5 = 20.

Syntax
```
value1 / value2
```

Parameters
value1 int or float
value2 int or float, but not zero (it is not possible divide by zero)

Related * (multiply)
% (modulo)

Name

/= (divide assign)

Examples

```
int a = 12;
int b = 3;
a /= b;  // Sets 'a' to 4
```

Description Combines division with assignment. The expression **a /= b** is equivalent to **a = a / b**.

Syntax

```
value1 /= value2
```

Parameters

value1 int or float
value2 any numerical value the same datatype as value1

Related = (assign)
/ (divide)

Name

& (bitwise AND)

```
int a = 207;    // In binary: 11001111
int b = 61;     // In binary: 00111101
int c = a & b; // In binary: 00001101
println(c);     // Prints "13", the decimal equivalent to
00001101
```

Examples

```
color argb = color(204, 204, 51, 255);
// The sytax "& 0xFF" compares the binary
// representation of the two values and
// makes all but the last 8 bits into a 0.
// "0xFF" is 00000000000000000000000011111111
int a = argb >> 24 & 0xFF;
int r = argb >> 16 & 0xFF;
int g = argb >> 8 & 0xFF;
int b = argb & 0xFF;
fill(r, g, b, a);
rect(30, 20, 55, 55);
```

Description Compares each corresponding bit in the binary representation of the values. For each comparison two 1's yield 1, 1 and 0 yield 0, and two 0's yield 0. This is easy to see when we look at the binary representation of numbers

```
  11010110  // 214
& 01011100  // 92
  --------
  01010100  // 84
```

To see the binary representation of a number, use the **binary()** function with **println()**.

Syntax

```
value & value2
```

Parameters

value1 int, char, byte
value2 int, char, byte

Related | (bitwise OR)
binary()

Name

<< (left shift)

Examples

```
int m = 1 << 3;   // In binary: 1 to 1000
println(m);  // Prints "8"
int n = 1 << 8;   // In binary: 1 to 100000000
println(n);  // Prints "256"
int o = 2 << 3;   // In binary: 10 to 10000
println(o);  // Prints "16"
int p = 13 << 1;  // In binary: 1101 to 11010
println(p);  // Prints "26"
```

```
// Packs four 8 bit numbers into one 32 bit number
int a = 255;  // Binary: 00000000000000000000000011111111
int r = 204;  // Binary: 00000000000000000000000011001100
int g = 204;  // Binary: 00000000000000000000000011001100
int b = 51;   // Binary: 00000000000000000000000000110011
a = a << 24;  // Binary: 11111111000000000000000000000000
r = r << 16;  // Binary: 00000000110011000000000000000000
g = g << 8;   // Binary: 00000000000000001100110000000000

// Equivalent to "color argb = color(r, g, b, a)" but faster
color argb = a | r | g | b;
fill(argb);
rect(30, 20, 55, 55);
```

Description Shifts bits to the left. The number to the left of the operator is shifted the number of places specified by the number to the right. Each shift to the left doubles the number, therefore each left shift multiplies the original number by 2. Use the left shift for fast multiplication or to pack a group of numbers together into one larger number. Left shifting only works with integers or numbers which automatically convert to an integer such at byte and char.

Syntax `value << n`

Parameters **value** int: the value to shift
n int: the number of places to shift left

Related >> (right shift)

Name

>> (right shift)

Examples

```
int m = 8 >> 3;    // In binary: 1000 to 1
println(m);  // Prints "1"
int n = 256 >> 6;  // In binary: 100000000 to 100
println(n);  // Prints "4"
int o = 16 >> 3;   // In binary: 10000 to 10
println(o);  // Prints "2"
int p = 26 >> 1;   // In binary: 11010 to 1101
println(p);  // Prints "13"
```

```
// Using "right shift" as a faster technique than red(), green(),
and blue()
color argb = color(204, 204, 51, 255);
int a = (argb >> 24) & 0xFF;
int r = (argb >> 16) & 0xFF;  // Faster way of getting red(argb)
int g = (argb >> 8) & 0xFF;   // Faster way of getting
green(argb)
```

```
int b = argb & 0xFF;          // Faster way of getting blue(argb)
fill(r, g, b, a);
rect(30, 20, 55, 55);
```

Description Shifts bits to the right. The number to the left of the operator is shifted the number of places specified by the number to the right. Each shift to the right halves the number, therefore each left shift divides the original number by 2. Use the right shift for fast divisions or to extract an individual number from a packed number. Right shifting only works with integers or numbers which automatically convert to an integer such at byte and char.

Bit shifting is helpful when using the **color** data type. A right shift can extract red, green, blue, and alpha values from a color. A left shift can be used to quickly reassemble a color value (more quickly than the **color()** function).

Syntax `value >> n`

Parameters **value** int: the value to shift
n int: the number of places to shift right

Related << (left shift)

Name

| (bitwise OR)

```
int a = 205;   // In binary: 11001101
int b = 45;    // In binary: 00101101
int c = a | b; // In binary: 11101101
println(c);    // Prints "237", the decimal equivalent to
11101101
```

Examples

```
int a = 255 << 24; // Binary: 11111111000000000000000000000000
int r = 204 << 16; // Binary: 00000000110011000000000000000000
int g = 204 << 8;  // Binary  00000000000000001100110000000000
int b = 51;        // Binary: 00000000000000000000000000110011
// OR the values together:    11111111111001100110011000110011
color argb = a | r | g | b;
fill(argb);
rect(30, 20, 55, 55);
```

Description Compares each corresponding bit in the binary representation of the values. For each comparison two 1's yield 1, 1 and 0 yield 1, and two 0's yield 0. This is easy to see when we look at the binary representation of numbers

```
  11010110  // 214
| 01011100  // 92
  --------
  11011110  // 222
```

To see the binary representation of a number, use the **binary()** function with **println()**.

Syntax `value | value2`

Parameters **value1** int, char, byte
value2 int, char, byte

Related & (bitwise AND)
binary()

Name

abs()

Examples

```
int a = abs(153);       // Sets 'a' to 153
int b = abs(-15);       // Sets 'b' to 15
float c = abs(12.234); // Sets 'c' to 12.234
float d = abs(-9.23);  // Sets 'd' to 9.23
```

Description Calculates the absolute value (magnitude) of a number. The absolute value of a number is always positive.

Syntax `abs(n)`

Parameters n int, or float: number to compute

Returns float or int

Name

ceil()

Examples

```
float x = 8.22;
int a = ceil(x);  // Sets 'a' to 9
```

Description Calculates the closest int value that is greater than or equal to the value of the parameter. For example, **ceil(9.03)** returns the value 10.

Syntax `ceil(n)`

Parameters n float: number to round up

Returns int

Related floor()
round()

Name

constrain()

Examples

```
void draw()
{
  background(204);
  float mx = constrain(mouseX, 30, 70);
  rect(mx-10, 40, 20, 20);
}
```

Description Constrains a value to not exceed a maximum and minimum value.

Syntax `constrain(amt, low, high)`

Parameters **amt** int, or float: the value to constrain
low int, or float: minimum limit
high int, or float: maximum limit

Returns float or int

Related max()
min()

Name

dist()

Examples

```
// Sets the background gray value based on the distance
// of the mouse from the center of the screen
void draw() {
```

```
  noStroke();
  float d = dist(width/2, height/2, mouseX, mouseY);
  float maxDist = dist(0, 0, width/2, height/2);
  float gray = map(d, 0, maxDist, 0, 255);
  fill(gray);
  rect(0, 0, width, height);
}
```

Description Calculates the distance between two points.

Syntax
```
dist(x1, y1, x2, y2)
dist(x1, y1, z1, x2, y2, z2)
```

Parameters
- **x1** float: x-coordinate of the first point
- **y1** float: y-coordinate of the first point
- **z1** float: z-coordinate of the first point
- **x2** float: x-coordinate of the second point
- **y2** float: y-coordinate of the second point
- **z2** float: z-coordinate of the second point

Returns float

Name

exp()

Examples
```
float v1 = exp(1.0);
println(v1);  // Prints "2.7182817"
```

Description Returns Euler's number *e* (2.71828...) raised to the power of the **n** parameter.

Syntax
```
exp(n)
```

Parameters **n** float: exponent to raise

Returns float

Name

floor()

Examples
```
float x = 2.88;
int a = floor(x);  // Sets 'a' to 2
```

Description Calculates the closest int value that is less than or equal to the value of the parameter.

Syntax
```
floor(n)
```

Parameters **n** float: number to round down

Returns int

Related ceil()
round()

Name

lerp()

Examples
```
float a = 20;
float b = 80;
float c = lerp(a, b, .2);
float d = lerp(a, b, .5);
```

```
float e = lerp(a, b, .8);
beginShape(POINTS);
vertex(a, 50);
vertex(b, 50);
vertex(c, 50);
vertex(d, 50);
vertex(e, 50);
endShape();


int x1 = 15;
int y1 = 10;
int x2 = 80;
int y2 = 90;
line(x1, y1, x2, y2);
for (int i = 0; i <= 10; i++) {
  float x = lerp(x1, x2, i/10.0) + 10;
  float y = lerp(y1, y2, i/10.0);
  point(x, y);
}
```

Description Calculates a number between two numbers at a specific increment. The **amt** parameter is the amount to interpolate between the two values where 0.0 equal to the first point, 0.1 is very near the first point, 0.5 is half-way in between, etc. The lerp function is convenient for creating motion along a straight path and for drawing dotted lines.

Syntax `lerp(start, stop, amt)`

Parameters
start float: first value
stop float: second value
amt float: float between 0.0 and 1.0

Returns float

Related curvePoint()
bezierPoint()

Name

log()

Examples

```
void setup() {
  int i = 12;
  println(log(i));
  println(log10(i));
}

// Calculates the base-10 logarithm of a number
float log10 (int x) {
  return (log(x) / log(10));
}
```

Description Calculates the natural logarithm (the base-*e* logarithm) of a number. This function expects the **n** parameter to be a value greater than 0.0.

Syntax `log(n)`

Parameters **n** float: number greater than 0.0

Returns float

Name

mag()

```
float x1 = 20;
float x2 = 80;
float y1 = 30;
float y2 = 70;
```

Examples

```
line(0, 0, x1, y1);
println(mag(x1, y1));  // Prints "36.05551"
line(0, 0, x2, y1);
println(mag(x2, y1));  // Prints "85.44004"
line(0, 0, x1, y2);
println(mag(x1, y2));  // Prints "72.8011"
line(0, 0, x2, y2);
println(mag(x2, y2));  // Prints "106.30146"
```

Description Calculates the magnitude (or length) of a vector. A vector is a direction in space commonly used in computer graphics and linear algebra. Because it has no "start" position, the magnitude of a vector can be thought of as the distance from the coordinate 0,0 to its x,y value. Therefore, **mag()** is a shortcut for writing **dist(0, 0, x, y)**.

Syntax

```
mag(a, b)
mag(a, b, c)
```

Parameters

a float: first value
b float: second value
c float: third value

Returns float

Related dist()

Name

map()

```
size(200, 200);
float value = 25;
float m = map(value, 0, 100, 0, width);
ellipse(m, 200, 10, 10);
```

```
float value = 110;
float m = map(value, 0, 100, -20, -10);
println(m);  // Prints "-9.0"
```

Examples

```
void setup() {
  size(200, 200);
  noStroke();
}

void draw() {
  background(204);
  float  x1 = map(mouseX, 0, width, 50, 150);
  ellipse(x1, 75, 50, 50);
  float x2 = map(mouseX, 0, width, 0, 200);
  ellipse(x2, 125, 50, 50);
}
```

Description Re-maps a number from one range to another.

In the first example above, the number 25 is converted from a value in the range

of 0 to 100 into a value that ranges from the left edge of the window (0) to the right edge (width).

As shown in the second example, numbers outside of the range are not clamped to the minimum and maximum parameters values, because out-of-range values are often intentional and useful.

Syntax

```
map(value, start1, stop1, start2, stop2)
```

Parameters

value float: the incoming value to be converted
start1 float: lower bound of the value's current range
stop1 float: upper bound of the value's current range
start2 float: lower bound of the value's target range
stop2 float: upper bound of the value's target range

Returns float

Related norm()
lerp()

Name

max()

Examples

```
int a = max(5, 9);            // Sets 'a' to 9
int b = max(-4, -12);         // Sets 'b' to -4
float c = max(12.3, 230.24);  // Sets 'c' to 230.24
```

```
int[] values = { 9, -4, 362, 21 };  // Create an array of ints
int d = max(values);                // Sets 'd' to 362
```

Description Determines the largest value in a sequence of numbers, and then returns that value. **max()** accepts either two or three **float** or **int** values as parameters, or an array of any length.

Syntax

```
max(a, b)
max(a, b, c)
max(list)
```

Parameters

a float, or int: first number to compare
b float, or int: second number to compare
c float, or int: third number to compare
list float[], or int[]: array of numbers to compare

Returns int or float

Related min()

Name

min()

Examples

```
int d = min(5, 9);            // Sets 'd' to 5
int e = min(-4, -12);         // Sets 'e' to -12
float f = min(12.3, 230.24);  // Sets 'f' to 12.3
```

```
int[] values = { 5, 1, 2, -3 };  // Create an array of ints
int h = min(values);             // Sets 'h' to -3
```

Description Determines the smallest value in a sequence of numbers, and then returns that value. **min()** accepts either two or three **float** or **int** values as parameters, or an array of any length.

Syntax

```
min(a, b)
```

```
min(a, b, c)
min(list)
```

Parameters
- **a** int, or float: first number
- **b** int, or float: second number
- **c** int, or float: third number
- **list** float[], or int[]: array of numbers to compare

Returns float or int

Related max()

Name

norm()

Examples

```
float value = 20;
float n = norm(value, 0, 50);
println(n);  // Prints "0.4"
```

```
float value = -10;
float n = norm(value, 0, 100);
println(n);  // Prints "-0.1"
```

Description

Normalizes a number from another range into a value between 0 and 1. Identical to **map(value, low, high, 0, 1)**.

Numbers outside of the range are not clamped to 0 and 1, because out-of-range values are often intentional and useful. (See the second example above.)

Syntax

```
norm(value, start, stop)
```

Parameters
- **value** float: the incoming value to be converted
- **start** float: lower bound of the value's current range
- **stop** float: upper bound of the value's current range

Returns float

Related map()
lerp()

Name

pow()

Examples

```
float a = pow( 1, 3);  // Sets 'a' to 1*1*1 = 1
float b = pow( 3, 5);  // Sets 'b' to 3*3*3*3*3 = 243
float c = pow( 3,-5);  // Sets 'c' to 1 / 3*3*3*3*3 = 1 / 243 = .0041
float d = pow(-3, 5);  // Sets 'd' to -3*-3*-3*-3*-3 = -243
```

Description

Facilitates exponential expressions. The **pow()** function is an efficient way of multiplying numbers by themselves (or their reciprocals) in large quantities. For example, **pow(3, 5)** is equivalent to the expression 3*3*3*3*3 and **pow(3, -5)** is equivalent to 1 / 3*3*3*3*3.

Syntax

```
pow(n, e)
```

Parameters
- **n** float: base of the exponential expression
- **e** float: power by which to raise the base

Returns float

Related sqrt()

Name

round()

Examples

```
float x = 9.2;
int rx = round(x);  // Sets 'rx' to 9

float y = 9.5;
int ry = round(y);  // Sets 'ry' to 10

float z = 9.9;
int rz = round(z);  // Sets 'rz' to 10
```

Description Calculates the integer closest to the **n** parameter. For example, **round(133.8)** returns the value 134.

Syntax `round(n)`

Parameters n float: number to round

Returns int

Related floor()
ceil()

Name

sq()

Examples

```
noStroke();
float a = sq(1);   // Sets 'a' to 1
float b = sq(-5);  // Sets 'b' to 25
float c = sq(9);   // Sets 'c' to 81
rect(0, 25, a, 10);
rect(0, 45, b, 10);
rect(0, 65, c, 10);
```

Description Squares a number (multiplies a number by itself). The result is always a positive number, as multiplying two negative numbers always yields a positive result. For example, -1 * -1 = 1.

Syntax `sq(n)`

Parameters n float: number to square

Returns float

Related sqrt()

Name

sqrt()

Examples

```
noStroke();
float a = sqrt(6561);  // Sets 'a' to 81
float b = sqrt(625);   // Sets 'b' to 25
float c = sqrt(1);     // Sets 'c' to 1
rect(0, 25, a, 10);
rect(0, 45, b, 10);
rect(0, 65, c, 10);
```

Description Calculates the square root of a number. The square root of a number is always

positive, even though there may be a valid negative root. The square root **s** of number **a** is such that **s*s = a**. It is the opposite of squaring.

Syntax `sqrt(n)`

Parameters **n** float: non-negative number

Returns float

Related pow()
sq()

Name

acos()

Examples

```
float a = PI;
float c = cos(a);
float ac = acos(c);
// Prints "3.1415927 : -1.0 : 3.1415927"
println(a + " : " + c + " : " +  ac);
```

```
float a = PI + PI/4.0;
float c = cos(a);
float ac = acos(c);
// Prints "3.926991 : -0.70710665 : 2.3561943"
println(a + " : " + c + " : " +  ac);
```

Description The inverse of **cos()**, returns the arc cosine of a value. This function expects the values in the range of -1 to 1 and values are returned in the range **0** to **PI (3.1415927)**.

Syntax `acos(value)`

Parameters **value** float: the value whose arc cosine is to be returned

Returns float

Related cos()
asin()
atan()

Name

asin()

Examples

```
float a = PI/3;
float s = sin(a);
float as = asin(s);
// Prints "1.0471976 : 0.86602545 : 1.0471976"
println(a + " : " + s + " : " +  as);
```

```
float a = PI + PI/3.0;
float s = sin(a);
float as = asin(s);
// Prints "4.1887903 : -0.86602545 : -1.0471976"
println(a + " : " + s + " : " +  as);
```

Description The inverse of **sin()**, returns the arc sine of a value. This function expects the values in the range of -1 to 1 and values are returned in the range **-PI/2** to **PI/2**.

Syntax `asin(value)`

Parameters **value** float: the value whose arc sine is to be returned

Returns float

Related sin()
acos()

atan()

Name

atan()

Examples

```
float a = PI/3;
float t = tan(a);
float at = atan(t);
// Prints "1.0471976 : 1.7320509 : 1.0471976"
println(a + " : " + t + " : " +  at);
```

```
float a = PI + PI/3.0;
float t = tan(a);
float at = atan(t);
// Prints "4.1887903 : 1.7320513 : 1.0471977"
println(a + " : " + t + " : " +  at);
```

Description The inverse of **tan()**, returns the arc tangent of a value. This function expects the values in the range of -Infinity to Infinity (exclusive) and values are returned in the range **-PI/2** to **PI/2** .

Syntax `atan(value)`

Parameters **value** float: -Infinity to Infinity (exclusive)

Returns float

Related tan()
asin()
acos()

Name

atan2()

Examples

```
void draw() {
  background(204);
  translate(width/2, height/2);
  float a = atan2(mouseY-height/2, mouseX-width/2);
  rotate(a);
  rect(-30, -5, 60, 10);
}
```

Description Calculates the angle (in radians) from a specified point to the coordinate origin as measured from the positive x-axis. Values are returned as a **float** in the range from **PI** to **-PI**. The **atan2()** function is most often used for orienting geometry to the position of the cursor. Note: The y-coordinate of the point is the first parameter, and the x-coordinate is the second parameter, due the the structure of calculating the tangent.

Syntax `atan2(y, x)`

Parameters **y** float: y-coordinate of the point
x float: x-coordinate of the point

Returns float

Related tan()

Name

cos()

Examples

```
float a = 0.0;
float inc = TWO_PI/25.0;
for (int i = 0; i < 25; i++) {
  line(i*4, 50, i*4, 50+cos(a)*40.0);
  a = a + inc;
}
```

Description Calculates the cosine of an angle. This function expects the values of the **angle** parameter to be provided in radians (values from 0 to PI*2). Values are returned in the range -1 to 1.

Syntax `cos(angle)`

Parameters **angle** float: an angle in radians

Returns float

Related sin()
tan()
radians()

Name

degrees()

Examples

```
float rad = PI/4;
float deg = degrees(rad);
println(rad + " radians is " + deg + " degrees");
```

Description Converts a radian measurement to its corresponding value in degrees. Radians and degrees are two ways of measuring the same thing. There are 360 degrees in a circle and 2*PI radians in a circle. For example, 90° = PI/2 = 1.5707964. All trigonometric functions in Processing require their parameters to be specified in radians.

Syntax `degrees(radians)`

Parameters **radians** float: radian value to convert to degrees

Returns float

Related radians()

Name

radians()

Examples

```
float deg = 45.0;
float rad = radians(deg);
println(deg + " degrees is " + rad + " radians");
```

Description Converts a degree measurement to its corresponding value in radians. Radians and degrees are two ways of measuring the same thing. There are 360 degrees in a circle and 2*PI radians in a circle. For example, 90° = PI/2 = 1.5707964. All trigonometric functions in Processing require their parameters to be specified in radians.

Syntax `radians(degrees)`

Parameters **degrees** float: degree value to convert to radians

Returns float

Related degrees()

Name

sin()

Examples

```
float a = 0.0;
float inc = TWO_PI/25.0;

for (int i = 0; i < 100; i=i+4) {
  line(i, 50, i, 50+sin(a)*40.0);
  a = a + inc;
}
```

Description Calculates the sine of an angle. This function expects the values of the **angle** parameter to be provided in radians (values from 0 to 6.28). Values are returned in the range -1 to 1.

Syntax `sin(angle)`

Parameters **angle** float: an angle in radians

Returns float

Related cos()
tan()
radians()

Name

tan()

Examples

```
float a = 0.0;
float inc = TWO_PI/50.0;

for (int i = 0; i < 100; i = i+2) {
  line(i, 50, i, 50+tan(a)*2.0);
  a = a + inc;
}
```

Description Calculates the ratio of the sine and cosine of an angle. This function expects the values of the **angle** parameter to be provided in radians (values from 0 to PI*2). Values are returned in the range **infinity** to **-infinity**.

Syntax `tan(angle)`

Parameters **angle** float: an angle in radians

Returns float

Related cos()
sin()
radians()

Name

noise()

Examples

```
float xoff = 0.0;

void draw() {
  background(204);
  xoff = xoff + .01;
```

```
  float n = noise(xoff) * width;
  line(n, 0, n, height);
}
```

```
float noiseScale=0.02;

void draw() {
  background(0);
  for (int x=0; x < width; x++) {
    float noiseVal = noise((mouseX+x)*noiseScale,
                           mouseY*noiseScale);
    stroke(noiseVal*255);
    line(x, mouseY+noiseVal*80, x, height);
  }
}
```

Description

Returns the Perlin noise value at specified coordinates. Perlin noise is a random sequence generator producing a more natural, harmonic succession of numbers than that of the standard **random()** function. It was invented by Ken Perlin in the 1980s and has been used in graphical applications to generate procedural textures, shapes, terrains, and other seemingly organic forms.

In contrast to the **random()** function, Perlin noise is defined in an infinite n-dimensional space, in which each pair of coordinates corresponds to a fixed semi-random value (fixed only for the lifespan of the program). The resulting value will always be between 0.0 and 1.0. Processing can compute 1D, 2D and 3D noise, depending on the number of coordinates given. The noise value can be animated by moving through the noise space, as demonstrated in the first example above. The 2nd and 3rd dimensions can also be interpreted as time.

The actual noise structure is similar to that of an audio signal, in respect to the function's use of frequencies. Similar to the concept of harmonics in physics, Perlin noise is computed over several octaves which are added together for the final result.

Another way to adjust the character of the resulting sequence is the scale of the input coordinates. As the function works within an infinite space, the value of the coordinates doesn't matter as such; only the *distance* between successive coordinates is important (such as when using **noise()** within a loop). As a general rule, the smaller the difference between coordinates, the smoother the resulting noise sequence. Steps of 0.005-0.03 work best for most applications, but this will differ depending on use.

Syntax

```
noise(x)
noise(x, y)
noise(x, y, z)
```

Parameters

x float: x-coordinate in noise space
y float: y-coordinate in noise space
z float: z-coordinate in noise space

Returns float

Related noiseSeed()
noiseDetail()
random()

Name

noiseDetail()

Examples

```
float noiseVal;
float noiseScale=0.02;

void draw() {
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width/2; x++) {
      noiseDetail(3,0.5);
      noiseVal = noise((mouseX+x) * noiseScale, (mouseY+y) *
noiseScale);
      stroke(noiseVal*255);
      point(x,y);
      noiseDetail(8,0.65);
      noiseVal = noise((mouseX + x + width/2) * noiseScale,
                       (mouseY + y) * noiseScale);
      stroke(noiseVal * 255);
      point(x + width/2, y);
    }
  }
}
```

Description Adjusts the character and level of detail produced by the Perlin noise function. Similar to harmonics in physics, noise is computed over several octaves. Lower octaves contribute more to the output signal and as such define the overal intensity of the noise, whereas higher octaves create finer-grained details in the noise sequence.

By default, noise is computed over 4 octaves with each octave contributing exactly half than its predecessor, starting at 50% strength for the first octave. This falloff amount can be changed by adding an additional function parameter. For example, a falloff factor of 0.75 means each octave will now have 75% impact (25% less) of the previous lower octave. While any number between 0.0 and 1.0 is valid, note that values greater than 0.5 may result in **noise()** returning values greater than 1.0.

By changing these parameters, the signal created by the **noise()** function can be adapted to fit very specific needs and characteristics.

Syntax

```
noiseDetail(lod)
noiseDetail(lod, falloff)
```

Parameters **lod** int: number of octaves to be used by the noise
falloff float: falloff factor for each octave

Returns void

Related noise()

Name

noiseSeed()

Examples

```
float xoff = 0.0;

void setup() {
  noiseSeed(0);
  stroke(0, 10);
}
```

```
void draw() {
  xoff = xoff + .01;
  float n = noise(xoff) * width;
  line(n, 0, n, height);
}
```

Description Sets the seed value for **noise()**. By default, **noise()** produces different results each time the program is run. Set the **seed** parameter to a constant to return the same pseudo-random numbers each time the software is run.

Syntax
```
noiseSeed(seed)
```

Parameters **seed** int: seed value

Returns void

Related noise()
noiseDetail()
random()
randomSeed()

Name

random()

```
for (int i = 0; i < 100; i++) {
  float r = random(50);
  stroke(r*5);
  line(50, i, 50+r, i);
}
```

Examples

```
for (int i = 0; i < 100; i++) {
  float r = random(-50, 50);
  println(r);
}
```

```
// Get a random element from an array
String[] words = { "apple", "bear", "cat", "dog" };
int index = int(random(words.length));  // Same as int(random(4))
println(words[index]);  // Prints one of the four words
```

Description Generates random numbers. Each time the **random()** function is called, it returns an unexpected value within the specified range. If only one parameter is passed to the function, it will return a float between zero and the value of the **high** parameter. For example, **random(5)** returns values between 0 and 5 (starting at zero, and up to, but not including, 5).

If two parameters are specified, the function will return a float with a value between the two values. For example, **random(-5, 10.2)** returns values starting at -5 and up to (but not including) 10.2. To convert a floating-point random number to an integer, use the **int()** function.

Syntax
```
random(high)
random(low, high)
```

Parameters **low** float: lower limit
high float: upper limit

Returns float

Related randomSeed()
noise()

Name

randomGaussian()

Examples

```
for (int y = 0; y < 100; i++) {
  float x = randomGaussian() * 15;
  line(50, y, 50 + x, y);
}
```

```
float[] distribution = new float[360];

void setup() {
  size(100, 100);
  for (int i = 0; i < distribution.length; i++) {
    distribution[i] = int(randomGaussian() * 15);
  }
}

void draw() {
  background(204);

  translate(width/2, width/2);

  for (int i = 0; i < distribution.length; i++) {
    rotate(TWO_PI/distribution.length);
    stroke(0);
    float dist = abs(distribution[i]);
    line(0, 0, dist, 0);
  }
}
```

Description Returns a float from a random series of numbers having a mean of 0 and standard deviation of 1. Each time the **randomGaussian()** function is called, it returns a number fitting a Gaussian, or normal, distribution. There is theoretically no minimum or maximum value that **randomGaussian()** might return. Rather, there is just a very low probability that values far from the mean will be returned; and a higher probability that numbers near the mean will be returned.

Syntax `randomGaussian()`

Returns float

Related random()
noise()

Name

randomSeed()

Examples

```
randomSeed(0);
for (int i=0; i < 100; i++) {
  float r = random(0, 255);
  stroke(r);
  line(i, 0, i, 100);
}
```

Description Sets the seed value for **random()**. By default, **random()** produces different results each time the program is run. Set the **seed** parameter to a constant to

return the same pseudo-random numbers each time the software is run.

Syntax `randomSeed(seed)`

Parameters **seed** int: seed value

Returns void

Related random()
noise()
noiseSeed()

Name

HALF_PI

Examples

```
float x = width/2;
float y = height/2;
float d = width * 0.8;
arc(x, y, d, d, 0, QUARTER_PI);
arc(x, y, d-20, d-20, 0, HALF_PI);
arc(x, y, d-40, d-40, 0, PI);
arc(x, y, d-60, d-60, 0, TWO_PI);
```

Description HALF_PI is a mathematical constant with the value 1.57079632679489661923. It is half the ratio of the circumference of a circle to its diameter. It is useful in combination with the trigonometric functions **sin()** and **cos()**.

Related PI
TWO_PI
TAU
QUARTER_PI

Name

PI

Examples

```
float x = width/2;
float y = height/2;
float d = width * 0.8;
arc(x, y, d, d, 0, QUARTER_PI);
arc(x, y, d-20, d-20, 0, HALF_PI);
arc(x, y, d-40, d-40, 0, PI);
arc(x, y, d-60, d-60, 0, TWO_PI);
```

Description PI is a mathematical constant with the value 3.14159265358979323846. It is the ratio of the circumference of a circle to its diameter. It is useful in combination with the trigonometric functions **sin()** and **cos()**.

Related TWO_PI
TAU
HALF_PI
QUARTER_PI

Name

QUARTER_PI

Examples

```
float x = width/2;
float y = height/2;
float d = width * 0.8;
arc(x, y, d, d, 0, QUARTER_PI);
arc(x, y, d-20, d-20, 0, HALF_PI);
arc(x, y, d-40, d-40, 0, PI);
arc(x, y, d-60, d-60, 0, TWO_PI);
```

Description QUARTER_PI is a mathematical constant with the value 0.7853982. It is one quarter the ratio of the circumference of a circle to its diameter. It is useful in combination with the trigonometric functions **sin()** and **cos()**.

Related
PI
TWO_PI
TAU
HALF_PI

Name

TAU

Examples

```
float x = width/2;
float y = height/2;
float d = width * 0.8;
arc(x, y, d, d, 0, QUARTER_PI);
arc(x, y, d-20, d-20, 0, HALF_PI);
arc(x, y, d-40, d-40, 0, PI);
arc(x, y, d-60, d-60, 0, TAU);
```

Description TAU is an alias for TWO_PI, a mathematical constant with the value 6.28318530717958647693. It is twice the ratio of the circumference of a circle to its diameter. It is useful in combination with the trigonometric functions **sin()** and **cos()**.

Related
PI
TWO_PI
HALF_PI
QUARTER_PI

Name

TWO_PI

Examples

```
float x = width/2;
float y = height/2;
float d = width * 0.8;
arc(x, y, d, d, 0, QUARTER_PI);
arc(x, y, d-20, d-20, 0, HALF_PI);
arc(x, y, d-40, d-40, 0, PI);
arc(x, y, d-60, d-60, 0, TWO_PI);
```

Description TWO_PI is a mathematical constant with the value 6.28318530717958647693. It is twice the ratio of the circumference of a circle to its diameter. It is useful in combination with the trigonometric functions **sin()** and **cos()**.

Related
PI

TAU
HALF_PI
QUARTER_PI
