Faculty Workshop

**Computational Art & Creative Coding  
Teaching CS1 with processing**

Presenters:

**Aaron Cadle**

**Ira Greenberg**

**Deepak Kumar**

**Ursula Wolz**

**Dianna Xu**

**July 9-12, 2014**

**Bryn Mawr College, Bryn Mawr, PA**

Table of Contents

[Getting Started. Welcome to Processing! 6](#_Toc313010860)

[Download Processing 6](#_Toc313010861)

[Your First Program 7](#_Toc313010862)

[Show 8](#_Toc313010863)

[Save 9](#_Toc313010864)

[Share 10](#_Toc313010865)

[Examples and Reference 11](#_Toc313010866)

[Processing Overview 13](#_Toc313010867)

[Sketching with Processing 14](#_Toc313010868)

[Hello world 14](#_Toc313010869)

[Hello mouse 15](#_Toc313010870)

[Exporting and distributing your work 16](#_Toc313010871)

[Creating images from your work 16](#_Toc313010872)

[Examples and reference 17](#_Toc313010873)

[More about size() 17](#_Toc313010874)

[Loading and displaying data 18](#_Toc313010875)

[Libraries add new features 19](#_Toc313010876)

[Sketching and scripting 20](#_Toc313010877)

[Don't start by trying to build a cathedral 20](#_Toc313010878)

[Coordinate System and Shapes 22](#_Toc313010879)

[Coordinate Space 22](#_Toc313010880)

[Simple Shapes 23](#_Toc313010881)

[Color 28](#_Toc313010882)

[Grayscale Color 28](#_Toc313010883)

[RGB Color 29](#_Toc313010884)

[Color Transparency 30](#_Toc313010885)

[Custom Color Ranges 31](#_Toc313010886)

[Drawing Curves 33](#_Toc313010887)

[Arcs 33](#_Toc313010888)

[Spline Curves 34](#_Toc313010889)

[Continuous Spline Curves 35](#_Toc313010890)

[Bézier Curves 36](#_Toc313010891)

[Continuous Bézier Curves 37](#_Toc313010892)

[Summary 38](#_Toc313010893)

[Strings and Drawing Text 39](#_Toc313010894)

[The String class 39](#_Toc313010895)

[Where do we find documentation for the String class? 39](#_Toc313010896)

[What is a String? 39](#_Toc313010897)

[Displaying Text 41](#_Toc313010898)

[Animating Text 43](#_Toc313010899)

[Rotating text 46](#_Toc313010900)

[Displaying text character by character 46](#_Toc313010901)

[Images and Pixels 53](#_Toc313010902)

[Getting started with images 53](#_Toc313010903)

[Your very first image processing filter 55](#_Toc313010904)

[Pixels, pixels, and more pixels 57](#_Toc313010905)

[Intro To Image Processing 60](#_Toc313010906)

[Our second image filter, making our own "tint" 61](#_Toc313010907)

[Writing to another PImage object's pixels 62](#_Toc313010908)

[Level II: Pixel Group Processing 64](#_Toc313010909)

[Visualizing the Image 67](#_Toc313010910)

[2-D Transformations 70](#_Toc313010911)

[Translation: Moving the Grid 70](#_Toc313010912)

[What’s the Advantage? 72](#_Toc313010913)

[Rotation 73](#_Toc313010914)

[Rotating the Correct Way 74](#_Toc313010915)

[Scaling 76](#_Toc313010916)

[Order Matters 77](#_Toc313010917)

[The Transformation Matrix 78](#_Toc313010918)

[Push and Pop 78](#_Toc313010919)

[Three-dimensional Transforms 78](#_Toc313010920)

[Case Study: An Arm-Waving Robot 78](#_Toc313010921)

[Case Study: Interactive Rotation 82](#_Toc313010922)

[Object Oriented Programming 85](#_Toc313010923)

[I'm Down with OOP 85](#_Toc313010924)

[Using an Object 86](#_Toc313010925)

[Writing the Cookie Cutter 88](#_Toc313010926)

[Using an Object: The Details 90](#_Toc313010927)

[Constructor Arguments 91](#_Toc313010928)

[Objects are data types too! 94](#_Toc313010929)

[Upcoming changes in Processing 2.0 (revision 0198+) 96](#_Toc313010930)

[Versions 96](#_Toc313010931)

[The Big Stuff 96](#_Toc313010932)

[Added and Improved 97](#_Toc313010933)

[Change and Removed 97](#_Toc313010934)

[What's Coming 97](#_Toc313010935)

# Getting Started. Welcome to Processing!

|  |  |
| --- | --- |
| This tutorial is for Processing 2+. If you see any errors or have comments, please [*let us know*](http://code.google.com/p/processing/issues/list). This tutorial was adapted from the book, [*Getting Started with Processing*](http://www.processing.org/learning/books/#reasfry2), by Casey Reas and Ben Fry, O'Reilly / Make 2010. *Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.* Download Processing Start by visiting <http://processing.org/download> and selecting the Mac, Windows, or Linux version, depending on what machine you have. Installation on each machine is straightforward:   * On Windows, you'll have a .zip file. Double-click it, and drag the folder inside to a location on your hard disk. It could be Program Files or simply the desktop, but the important thing is for the processing folder to be pulled out of that .zip file. Then double-click processing.exe to start. * The Mac OS X version is a disk image (.dmg) file. Drag the Processing icon to the Applications folder. If you're using someone else's machine and can't modify the Applications folder, just drag the application to the desktop. Then double-click the Processing icon to start. * The Linux version is a .tar.gz file, which should be familiar to most Linux users. Download the file to your home directory, then open a terminal window, and type: tar xvfz processing-xxxx.tgz (Replace xxxx with the rest of the file's name, which is the version number.) This will create a folder named processing-2.0 or something similar. Then change to that directory: cd processing-xxxx and run it: ./processing   With any luck, the main Processing window will now be visible. Everyone's setup is different, so if the program didn't start, or you're otherwise stuck, visit the [troubleshooting page](http://wiki.processing.org/w/Troubleshooting) for possible solutions.    PDE The Processing Development Environment. Your First Program You're now running the Processing Development Environment (or PDE). There's not much to it; the large area is the Text Editor, and there's a row of buttons across the top; this is the toolbar. Below the editor is the Message Area, and below that is the Console. The Message Area is used for one line messages, and the Console is used for more technical details.  In the editor, type the following:  ellipse(50, 50, 80, 80);  This line of code means "draw an ellipse, with the center 50 pixels over from the left and 50 pixels down from the top, with a width and height of 80 pixels." Click the Run button, which looks like this:  Run  If you've typed everything correctly, you'll see this appear in the Display Window:  http://processing.org/learning/gettingstarted/imgs/Ex_02_01.gif  If you didn't type it correctly, the Message Area will turn red and complain about an error. If this happens, make sure that you've copied the example code exactly: the numbers should be contained within parentheses and have commas between each of them, and the line should end with a semicolon.  One of the most difficult things about getting started with programming is that you have to be very specific about the syntax. The Processing software isn't always smart enough to know what you mean, and can be quite fussy about the placement of punctuation. You'll get used to it with a little practice.  Next, we'll skip ahead to a sketch that's a little more exciting. Delete the text from the last example, and try this:  http://processing.org/learning/gettingstarted/imgs/Ex_02_02.gif  void **setup**() {    size(480, 120);  }  void **draw**() {    if (mousePressed) {      fill(0);    } else {      fill(255);    }    ellipse(mouseX, mouseY, 80, 80);  }  This program creates a window that is 480 pixels wide and 120 pixels high, and then starts drawing white circles at the position of the mouse. When a mouse button is pressed, the circle color changes to black. We'll explain more about the elements of this program in detail later. For now, run the code, move the mouse, and click to experience it. Show So far we've covered only the Run button, though you've probably guessed what the Stop button next to it does:  Stop  If you don't want to use the buttons, you can always use the Sketch menu, which reveals the shortcut Ctrl-R (or Cmd-R on the Mac) for Run. Below Run in the Sketch menu is Present, which clears the rest of the screen to present your sketch all by itself:  Sketch menu  You can also use Present from the toolbar by holding down the Shift key as you click the Run button. Save The next command that's important is Save. It's the downward arrow on the toolbar:  Save  You can also find it under the File menu. By default, your programs are saved to the "sketchbook," which is a folder that collects your programs for easy access. Clicking the Open button on the toolbar (the arrow pointing up) will bring up a list of all the sketches in your sketchbook, as well as a list of examples that are installed with the Processing software:  Open  It's always a good idea to save your sketches often. As you try different things, keep saving with different names, so that you can always go back to an earlier version. This is especially helpful if — no, when — something breaks. You can also see where the sketch is located on the disk with Show Sketch Folder under the Sketch menu.  You can also create a new sketch by pressing the New button on the toolbar:  New  This will replace the sketch in the current window with an empty one. Holding down Shift when you press the New button will create a new sketch in its own window, as will selecting File > New. The Open button works the same way. Share Another theme of Processing is sharing your work. The Export button on the toolbar:  Export  will bundle your code into an application for your choice of Mac, Windows, and/or Linux. This is an easy way to make self-contained, double-clickable versions of your projects.    Export to Application menu Export to Application menu.    You can also find Export, along with its sibling Export to Applet, underneath the File menu. Export to Applet bundles your code into a single folder titled applet that can be uploaded to a web server. After exporting, the applet folder will open on your desktop. The PDE file is the source code, the JAR file is the program, the HTML file is the web page, and the GIF file is displayed in the web browser while the program is loading. Double-clicking the index.html file will launch your web browser and show your sketch on the web page it has created.  Holding down Shift when you press the Export button on the toolbar is another way to use Export to Applet.    Applet folder The applet folder contains the exported sketch.     |  | | --- | | NOTE: The export folders are erased and recreated each time you use the Export command, so be sure to move the folder elsewhere before you make any changes to the HTML file or anything else inside. Alternatively, you can turn off the automatic file erasure in the Preferences. |   In addition to exporting your code as applications and applets, you can switch to a different mode within Processing to export to other platforms. For example, changing to [JavaScript Mode](http://processing.org/learning/javascript/) exports HTML5 Canvas and WebGL. Changing to [Android](http://processing.org/learning/android/) mode exports application for Android phones and tablets. Examples and Reference Learning how to program with Processing involves exploring lots of code: running, altering, breaking, and enhancing it until you have reshaped it into something new. With this in mind, the Processing software download includes dozens of examples that demonstrate different features of the software. To open an example, select Examples from the File menu or click the Open icon in the PDE. The examples are grouped into categories based on their function, such as Form, Motion, and Image. Find an interesting topic in the list and try an example.  If you see a part of the program you're unfamiliar with that is colored orange (this means it's a part of Processing), select its name, and then click on "Find in Reference" from the Help menu. You can also right-click the text (or Ctrl-click on a Mac) and choose Find in Reference from the menu that appears. This will open the reference for the selected code element in your web browser. [The reference is also available online](http://www.processing.org/reference/).  The Processing Reference explains every code element with a description and examples. The reference programs are much shorter (usually four or five lines) and easier to follow than the longer code found in the Examples folder. We recommend keeping the reference open while you're reading this book and while you're programming. It can be navigated by topic or alphabetically; sometimes it's fastest to do a text search within your browser window.  The reference was written with the beginner in mind; we hope that we've made it clear and understandable. We're grateful to the many people who've spotted errors over the years and reported them. If you think you can improve a reference entry or you find a mistake, please let us know by clicking on the link at the top of each reference page. |

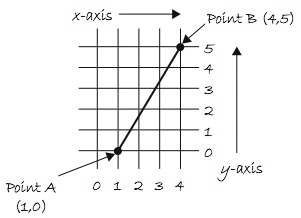
# Processing Overview

|  |  |  |
| --- | --- | --- |
| This tutorial is for Processing 2+. If you see any errors or have comments, please [*let us know*](http://code.google.com/p/processing/issues/list). This tutorial is adapted from the book, [*Visualizing Data*](http://www.processing.org/learning/books/#fry) by Ben Fry, O'Reilly 2007. *Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*  Processing is a simple programming environment that was created to make it easier to develop visually oriented applications with an emphasis on animation and providing users with instant feedback through interaction. The developers wanted a means to “sketch” ideas in code. As its capabilities have expanded over the past six years, Processing has come to be used for more advanced production-level work in addition to its sketching role. Originally built as a domain-specific extension to Java targeted towards artists and designers, Processing has evolved into a full-blown design and prototyping tool used for large-scale installation work, motion graphics, and complex data visualization.  Processing is based on Java, but because program elements in Processing are fairly simple, you can learn to use it even if you don't know any Java. If you're familiar with Java, it's best to forget that Processing has anything to do with Java for a while, until you get the hang of how the API works.  The latest version of Processing can be downloaded at [*http://processing.org/download*](http://processing.org/download)  An important goal for the project was to make this type of programming accessible to a wider audience. For this reason, Processing is free to download, free to use, and open source. But projects developed using the Processing environment and core libraries can be used for any purpose. This model is identical to GCC, the GNU Compiler Collection. GCC and its associated libraries (e.g. libc) are open source under the GNU Public License (GPL), which stipulates that changes to the code must be made available. However, programs created with GCC (examples too numerous to mention) are not themselves required to be open source.  Processing consists of:   * The Processing Development Environment (PDE). This is the software that runs when you double-click the Processing icon. The PDE is an Integrated Development Environment (IDE) with a minimalist set of features designed as a simple introduction to programming or for testing one-off ideas. * A collection of functions (also referred to as commands or methods) that make up the “core” programming interface, or API, as well as several libraries that support more advanced features such as sending data over a network, reading live images from a webcam, and saving complex imagery in PDF format. * A language syntax, identical to Java but with a few modifications. * An active online community, based at [*http://processing.org*](http://processing.org).   For this reason, references to “Processing” can be somewhat ambiguous. Are we talking about the API, the development environment, or the web site? We'll be careful in this text when referring to each. Sketching with Processing A Processing program is called a *sketch*. The idea is to make Java-style programming feel more like scripting, and adopt the process of scripting to quickly write code. Sketches are stored in the *sketchbook*, a folder that's used as the default location for saving all of your projects. When you first run Processing, the sketch last used will automatically open. If this is the first time Processing is used (or if the sketch is no longer available), a new sketch will open.  Sketches that are stored in the sketchbook can be accessed from File → Sketchbook. Alternatively, File → Open... can be used to open a sketch from elsewhere on the system.  Advanced programmers need not use the PDE, and may instead choose to use its libraries with the Java environment of choice. However, if you're just getting started, it's recommended that you use the PDE for your first few projects to gain familiarity with the way things are done. While Processing is based on Java, it was never meant to be a Java IDE with training wheels. To better address our target audience, the conceptual model (how programs work, how interfaces are built, and how files are handled) is somewhat different from Java. Hello world The Processing equivalent of a "Hello World" program is simply to draw a line:  line(15, 25, 70, 90);  Enter this example and press the Run button, which is an icon that looks like the Play button from any audio or video device. Your code will appear in a new window, with a gray background and a black line from coordinate (15, 25) to (70, 90). The (0, 0) coordinate is the upper left-hand corner of the display window. Building on this program to change the size of the display window and set the background color, type in the code below:  size(400, 400);  background(192, 64, 0);  stroke(255);  line(150, 25, 270, 350);  This version sets the window size to 400 x 400 pixels, sets the background to an orange-red, and draws the line in white, by setting the stroke color to 255. By default, colors are specified in the range 0 to 255. Other variations of the parameters to the stroke() function provide alternate results:  stroke(255); // sets the stroke color to white  stroke(255, 255, 255); // identical to the line above  stroke(255, 128, 0); // bright orange (red 255, green 128, blue 0)  stroke(#FF8000); // bright orange as a web color  stroke(255, 128, 0, 128); // bright orange with 50% transparency  The same alternatives work for the fill() function, which sets the fill color, and the background() function, which clears the display window. Like all Processing functions that affect drawing properties, the fill and stroke colors affect all geometry drawn to the screen until the next fill and stroke functions.   |  | | --- | | *Advanced Topic: External editor*  It's also possible to use the editor of your choice instead of the built-in editor. Simply select “Use External Editor” in the Preferences window (Processing → Preferences on Mac OS X, or File → Preferences on Windows and Linux). When using an external editor, editing will be disabled in the PDE, but the text will reload whenever you press Run. |  Hello mouse A program written as a list of statements (like the previous examples) is called a *static* sketch. In a static sketch, a series of functions are used to perform tasks or create a single image without any animation or interaction. Interactive programs are drawn as a series of frames, which you can create by adding functions titled setup() and draw() as shown in the code below. These are built-in functions that are called automatically.  void setup() {  size(400, 400);  stroke(255);  background(192, 64, 0);  }  void draw() {  line(150, 25, mouseX, mouseY);  }  The setup() block runs once, and the draw() block runs repeatedly. As such, setup() can be used for any initialization; in this case, setting the screen size, making the background orange, and setting the stroke color to white. The draw() block is used to handle animation. The size() function must always be the first line inside setup().  Because the background() function is used only once, the screen will fill with lines as the mouse is moved. To draw just a single line that follows the mouse, move the background() function to the draw() function, which will clear the display window (filling it with orange) each time draw() runs.  void setup() {  size(400, 400);  stroke(255);  }  void draw() {  background(192, 64, 0);  line(150, 25, mouseX, mouseY);  }  Static programs are most commonly used for extremely simple examples, or for scripts that run in a linear fashion and then exit. For instance, a static program might start, draw a page to a PDF file, and exit.  Most programs will use the setup() and draw() blocks. More advanced mouse handling can also be introduced; for instance, the mousePressed() function will be called whenever the mouse is pressed. In the following example, when the mouse is pressed, the screen is cleared via the background() function:  void setup() {  size(400, 400);  stroke(255);  }    void draw() {  line(150, 25, mouseX, mouseY);  }    void mousePressed() {  background(192, 64, 0);  }   Exporting and distributing your work One of the most significant features of the Processing environment is its ability to bundle your sketch into an application or applet with just one click. Select File → Export to package your current sketch as an application. This will to bundle your sketch as an application for Windows, Mac OS X, and Linux. Similarly, you can use File → Export Applet to create a Java Applet from your code. This will create a folder named applet inside your sketch folder. Opening the *index.html* file inside that folder will open your sketch in a browser. The applet folder can be copied to a web site intact, and will be viewable by anyone who has Java installed on their system.   The application and applet folders are overwritten whenever you export—make a copy or remove them from the sketch folder before making changes to the *index.html* file or the contents of the folder. Alternatively, you can turn off the automatic file erasure in the Preferences.  More about the export features can be found in the reference at [*http://processing.org/reference/environment/#Export*](http://processing.org/reference/environment/#Export) Creating images from your work If you don't want to distribute the actual project, you might want to create images of its output instead. Images are saved with the saveFrame() function. Adding saveFrame() at the end of draw() will produce a numbered sequence of TIFF-format images of the program's output, named *screen-0001.tif*, *screen-0002.tif*, and so on. A new file will be saved each time draw() runs — watch out, this can quickly fill your sketch folder with hundreds of files. You can also specify your own name and file type for the file to be saved with a function like:  saveFrame("output.png")  To do the same for a numbered sequence, use # (hash marks) where the numbers should be placed:  saveFrame("output-####.png");  For high quality output, you can write geometry to PDF files instead of the screen, as described in the later section about the size() function. Examples and reference While many programmers learn to code in school, others teach themselves and learn on their own. Learning on your own involves looking at lots of other code: running, altering, breaking, and enhancing it until you can reshape it into something new. With this learning model in mind, the Processing software download includes hundreds of examples that demonstrate different features of the environment and API.  The examples can be accessed from the File → Examples menu. They're grouped into categories based on their function (such as Motion, Typography, and Image) or the libraries they use (PDF, Network, and Video).   Find an interesting topic in the list and try an example. You'll see functions that are familiar, e.g. stroke(), line(), and background(), as well as others that have not yet been covered. To see how a function works, select its name, and then right-click and choose Find in Reference from the pop-up menu (Find in Reference can also be found beneath the Help menu). This will open the reference for that function in your default web browser.  In addition to a description of the function's syntax, each reference page includes an example that uses the function. The reference examples are much shorter (usually four or five lines apiece) and easier to follow than the longer code examples. More about size() The size() function sets the global variables width and height. For objects whose size is dependent on the screen, always use the width and height variables instead of a number. This prevents problems when the size() line is altered.  size(400, 400);  // The wrong way to specify the middle of the screen  ellipse(200, 200, 50, 50);  // Always the middle, no matter how the size() line changes  ellipse(width/2, height/2, 50, 50);  In the earlier examples, the size() function specified only a width and height for the window to be created. An optional parameter to the size() function specifies how graphics are rendered. A renderer handles how the Processing API is implemented for a particular output function (whether the screen, or a screen driven by a high-end graphics card, or a PDF file). Several renderers are included with Processing, each having a unique function. At the risk of getting too far into the specifics, here's a description of the possible drawing modes to use with Processing.  size(400, 400, P2D);  The P2D renderer is used by default, so this statement is identical to size(400, 400). The P2D renderer does an excellent job with high-quality 2D vector graphics, but at the expense of speed. In particular, working with pixels directly is slow.  size(400, 400, P3D);  The P3D renderer uses Sun's JOGL (Java for OpenGL) library for faster rendering, while using Processing's simpler graphics APIs and the Processing development environment's easy application export.  size(400, 400, PDF, "output.pdf");  The PDF renderer draws all geometry to a file instead of the screen. To use PDF, in addition to altering your size() function, you must select Import Library, then PDF from the Sketch menu. This is a cousin of the Java2D renderer, but instead writes directly to PDF files. Loading and displaying data One of the unique aspects of the Processing API is the way files are handled. The loadImage() and loadStrings() functions each expect to find a file inside a folder named *data*, which is a subdirectory of the sketch folder.   |  | | --- | | *Advanced Topic: Notes on the data folder*  The *data* folder addresses a common frustration when dealing with code that is tested locally but deployed over the web. Like Java, software written with Processing is subject to security restrictions that determine how a program can access resources such as the local hard disk or other servers via the Internet. This prevents malicious developers from writing code that could harm your computer or compromise your data.  The security restrictions can be tricky to work with during development. When running a program locally, data can be read directly from the disk, though it must be placed relative to the user's “working directory,” generally the location of the application. When running online, data must come from a location on the same server. It might be bundled with the code itself (in a JAR archive, discussed later; or from another URL on the same server). For a local file, Java's FileInputStream class can be used. If the file is bundled in a JAR archive, the getResource() function is used. For a file on the server, URL.openStream() might be employed. During the journey from development to deployment, it may be necessary to use all three of these functions.  With Processing, each of these scenarios (and some others) is handled transparently by the file API functions. By placing resources in the data folder, Processing packages the files as necessary for online and offline use. |   File handling functions include loadStrings(), which reads a text file into an array of String objects, and loadImage() which reads an image into a PImage object, the container for image data in Processing.  // Examples of loading a text file and a JPEG image  // from the data folder of a sketch.  String[] lines = loadStrings("something.txt");  PImage image = loadImage("picture.jpg");  These examples may be a bit easier to read if you know the programming concepts of data types and classes. Each variable has to have a data type, such as String or PImage.  The String[] syntax means “an array of data of the class String.” This array is created by the loadStrings function and is given the name lines; it will presumably be used later in the program under this name. The reason loadStrings creates an array is that it splits the *something.txt* file into its individual lines. The following function creates a single variable of class PImage, with the name image.  To add a file to the data folder of a Processing sketch, use the Sketch → Add File menu option, or drag the file into the editor window of the PDE. The data folder will be created if it does not exist already.   To view the contents of the sketch folder, use the Sketch → Show Sketch Folder menu option. This opens the sketch window in your operating system's file browser. Libraries add new features A *library* is a collection of code in a specified format that makes it easy to use within Processing. Libraries have been important to the growth of the project, because they let developers make new features accessible to users without needing to make them part of the core Processing API.  Several core libraries come with Processing. These can be seen in the Libraries section of the online reference (also available from the Help menu from within the PDE.) These libraries can be seen at [*http://processing.org/reference/libraries/*](http://processing.org/reference/libraries/)  One example is the PDF Export library. This library makes it possible to write PDF files directly from Processing. These vector graphics files can be scaled to any size and output at very high resolutions.  To use the PDF library in a project, choose Sketch → Import Library → pdf. This will add the following line to the top of the sketch:  import processing.pdf.\*;  Java programmers will recognize the import command. In Processing, this line is also used to determine what code is packaged with a sketch when it is exported as an applet or application.  Now that the PDF library is imported, you may use it to create a file. For instance, the following line of code creates a new PDF file named *lines.pdf* that you can draw to.  beginRecord(PDF, "line.pdf");  Each drawing function such as line() and ellipse() will now draw to the screen as well as to the PDF.  Other libraries provide features such as reading images from a camera, sending and receiving MIDI and OSC commands, sophisticated 3D camera control, and access to MySQL databases. Sketching and scripting Processing sketches are made up of one or more tabs, with each tab representing a piece of code. The environment is designed around projects that are a few pages of code, and often three to five tabs in total. This covers a significant number of projects developed to test and prototype ideas, often before embedding them into a larger project or building a more robust application for broader deployment.  The idea of sketching is identical to that of scripting, except that you're not working in an interpreted scripting language, but rather gaining the performance benefit of compiling to Java class files. Of course, strictly speaking, Java is itself an interpreted language, but its bytecode compilation brings it much closer to the "metal" than languages such as JavaScript, Python, or Ruby.  Processing was never intended as the ultimate language for programming visuals; instead, we set out to make something that was:   * A sketchbook for our own work, simplifying the majority of tasks that we undertake. * A programming environment suitable for teaching programming to a non-traditional audience. * A stepping stone from scripting languages to more complicated or difficult languages such as full-blown Java or C++.   At the intersection of these points is a tradeoff between speed and simplicity of use. If we didn't care about speed, it might make sense to use Python, Ruby, or many other scripting languages. This is especially true for the education side. If we didn't care about making a transition to more advanced languages, we'd probably avoid a C++ or Java-style syntax. But Java makes a nice starting point for a sketching language because it's far more forgiving than C/C++ and also allows users to export sketches for distribution via the web.   Processing assembles our experience in building software of this kind (sketches of interactive works or data-driven visualization) and simplifies the parts that we felt should be easier, such as getting started quickly, and insulating new users from issues like those associated with setting up Java. Don't start by trying to build a cathedral If you're already familiar with programming, it's important to understand how Processing differs from other development environments and languages. The Processing project encourages a style of work that builds code quickly, understanding that either the code will be used as a quick sketch, or ideas are being tested before developing a final project. This could be misconstrued as software engineering heresy. Perhaps we're not far from “hacking,” but this is more appropriate for the roles in which Processing is used. Why force students or casual programmers to learn about graphics contexts, threading, and event handling functions before they can show something on the screen that interacts with the mouse? The same goes for advanced developers: why should they always need to start with the same two pages of code whenever they begin a project?  In another scenario, the ability to try things out quickly is a far higher priority than sophisticated code structure. Usually you don't know what the outcome will be, so you might build something one week to try an initial hypothesis, and build something new the next based on what was learned in the first week. To this end, remember the following considerations as you begin writing code with Processing:   * Be careful about creating unnecessary structures in your code. As you learn about encapsulating your code into classes, it's tempting to make ever-smaller classes, because data can always be distilled further. Do you need classes at the level of molecules, atoms, or quarks? Just because atoms go smaller doesn't mean that we need to work at a lower level of abstraction. If a class is half a page, does it make sense to have six additional subclasses that are each half a page long? Could the same thing be accomplished with a single class that is a page and a half in total? * Consider the scale of the project. It's not always necessary to build enterprise-level software on the first day. Explore first: figure out the minimum code necessary to help answer your questions and satisfy your curiosity.   The argument is not to avoid continually rewriting, but rather to delay engineering work until it's appropriate. The threshold for where to begin engineering a piece of software is much later than for traditional programming projects because there is a kind of *art* to the early process of quick iteration.  Of course, once things are working, avoid the urge to rewrite for its own sake. A rewrite should be used when addressing a completely different problem. If you've managed to hit the nail on the head, you should refactor to clean up function names and class interactions. But a full rewrite of already finished code is almost always a bad idea, no matter how "ugly" it may seem. |

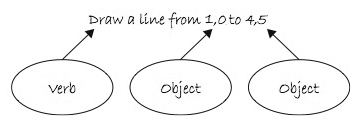
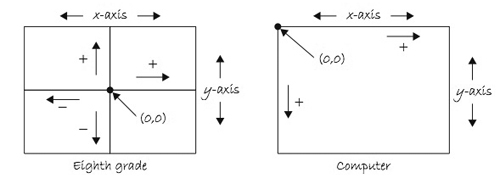
# Coordinate System and Shapes

*This tutorial is for Processing version 1.1+. If you see any errors or have comments, please* [*let us know*](http://code.google.com/p/processing/issues/list)*. This tutorials is from the book,* [*Learning Processing*](http://www.processing.org/learning/books/#shiffman)*, by Daniel Shiffman, published by Morgan Kaufmann Publishers, Copyright 2008 Elsevier Inc. All rights reserved.*  
*Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*

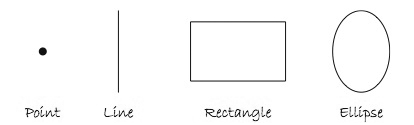
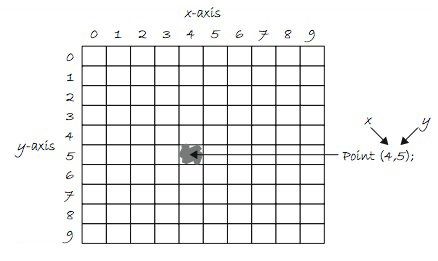
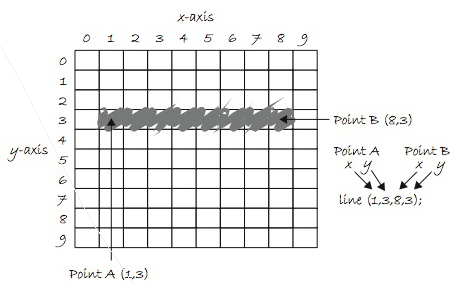
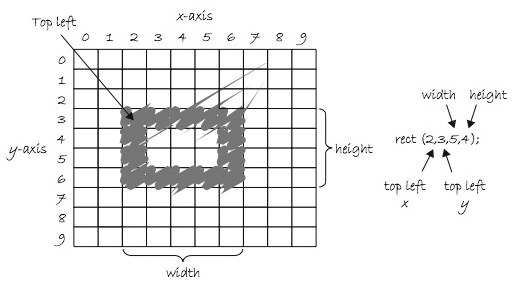
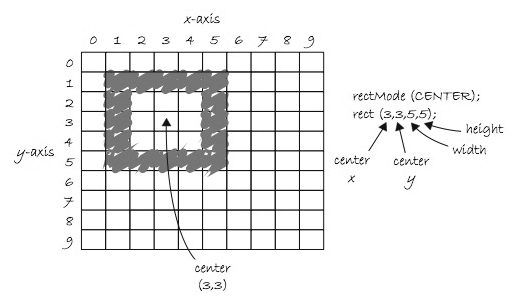
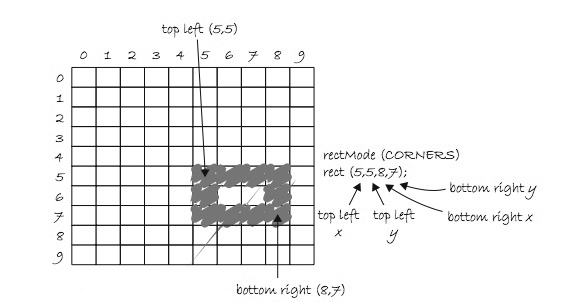
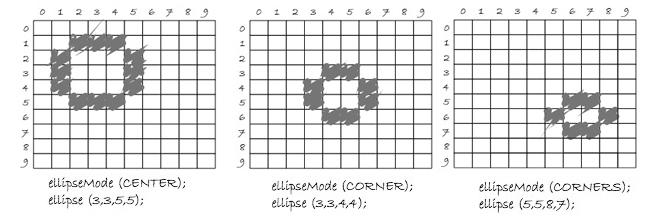
## Coordinate Space

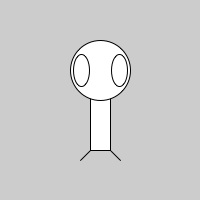
Before we begin programming with Processing, we must first channel our eighth grade selves, pull out a piece of graph paper, and draw a line. The shortest distance between two points is a good old fashioned line, and this is where we begin, with two points on that graph paper.   
  
  
  
The above figure shows a line between point A (1,0) and point B (4,5). If you wanted to direct a friend of yours to draw that same line, you would give them a shout and say "draw a line from the point one-zero to the point four-five, please." Well, for the moment, imagine your friend was a computer and you wanted to instruct this digital pal to display that same line on its screen. The same command applies (only this time you can skip the pleasantries and you will be required to employ a precise formatting). Here, the instruction will look like this:

line(1,0,4,5);

Even without having studied the syntax of writing code, the above statement should make a fair amount of sense. We are providing a *command* (which we will refer to as a "function") for the machine to follow entitled "line." In addition, we are specifying some arguments for how that line should be drawn, from point A (1,0) to point B (4,5). If you think of that line of code as a sentence, the function is a verb and the arguments are the objects of the sentence. The code sentence also ends with a semicolon instead of a period.   
  
  
  
The key here is to realize that the computer screen is nothing more than a fancier piece of graph paper. Each pixel of the screen is a coordinate - two numbers, an "x" (horizontal) and a "y" (vertical) - that determines the location of a point in space. And it is our job to specify what shapes and colors should appear at these pixel coordinates.   
  
Nevertheless, there is a catch here. The graph paper from eighth grade ("Cartesian coordinate system") placed (0,0) in the center with the y-axis pointing up and the x-axis pointing to the right (in the positive direction, negative down and to the left). Th e coordinate system for pixels in a computer window, however, is reversed along the y-axis. (0,0) can be found at the top left with the positive direction to the right horizontally and down vertically.   
  


## Simple Shapes

The vast majority of the programming examples you'll see with Processing are visual in nature. These examples, at their core, involve drawing shapes and setting pixels. Let's begin by looking at four primitive shapes.   
  
  
  
For each shape, we will ask ourselves what information is required to specify the location and size (and later color) of that shape and learn how Processing expects to receive that information. In each of the diagrams below, we'll assume a window with a width of 10 pixels and height of 10 pixels. This isn't particularly realistic since when you really start coding you will most likely work with much larger windows (10x10 pixels is barely a few millimeters of screen space). Nevertheless for demonstration purposes, it is nice to work with smaller numbers in order to present the pixels as they might appear on graph paper (for now) to better illustrate the inner workings of each line of code.   
  
A [**point()**](http://www.processing.org/reference/point_.html) is the easiest of the shapes and a good place to start. To draw a point, we only need an x and y coordinate.   
  
  
  
A [**line()**](http://www.processing.org/reference/line_.html) isn't terribly difficult either and simply requires two points: (x1,y1) and (x2,y2):   
  
  
  
Once we arrive at drawing a [**rect()**](http://www.processing.org/reference/rect_.html), things become a bit more complicated. In Processing, a rectangle is specified by the coordinate for the top left corner of the rectangle, as well as its width and height.   
  
  
  
A second way to draw a rectangle involves specifying the centerpoint, along with width and height. If we prefer this method, we first indicate that we want to use the "CENTER" mode before the instruction for the rectangle itself. Note that Processing is case-sensitive.   
  
  
  
Finally, we can also draw a rectangle with two points (the top left corner and the bottom right corner). The mode here is "CORNERS".   
  
  
  
Once we have become comfortable with the concept of drawing a rectangle, an [**ellipse()**](http://www.processing.org/reference/ellipse_.html) is a snap. In fact, it is identical to **rect()** with the difference being that an ellipse is drawn where the bounding box of the rectangle would be. The default mode for *ellipse()* is "CENTER", rather than "CORNER."   
  
  
It is important to acknowledge that these ellipses do not look particularly circular. Processing has a built-in methodology for selecting which pixels should be used to create a circular shape. Zoomed in like this, we get a bunch of squares in a circle-like pattern, but zoomed out on a computer screen, we get a nice round ellipse. Processing also gives us the power to develop our own algorithms for coloring in individual pixels (in fact, we can already imagine how we might do this using "point" over and over again), but for now, we are content with allowing the "ellipse" statement to do the hard work. (For more about pixels, start with: [the pixels reference page](http://processing.org/reference/pixels.html), though be warned this is a great deal more advanced than this tutorial.)

Now let's look at what some code with shapes in more realistic setting, with window dimensions of 200 by 200. Note the use of the [size()](http://www.processing.org/reference/size_.html) function to specify the width and height of the window.   
  


size(200,200);

rectMode(CENTER);

rect(100,100,20,100);

ellipse(100,70,60,60);

ellipse(81,70,16,32);

ellipse(119,70,16,32);

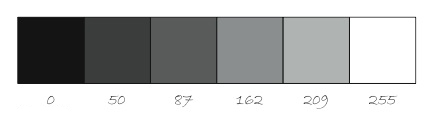
line(90,150,80,160);

line(110,150,120,160);

# Color

*This tutorial is for Processing version 1.1+. If you see any errors or have comments, please* [*let us know*](http://code.google.com/p/processing/issues/list)*. This tutorials is from the book,* [*Learning Processing*](http://www.processing.org/learning/books/#shiffman)*, by Daniel Shiffman, published by Morgan Kaufmann Publishers, Copyright 2008 Elsevier Inc. All rights reserved.  
Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*

## Grayscale Color

In the digital world, when we want to talk about a color, precision is required. Saying "Hey, can you make that circle bluish-green?" will not do. Color, rather, is defined as a range of numbers. Let's start with the simplest case: black & white or grayscale. 0 means black, 255 means white. In between, every other number - 50, 87, 162, 209, and so on - is a shade of gray ranging from black to white.   
  


|  |
| --- |
|  |

|  |
| --- |
| *Does 0-255 seem arbitary to you?* Color for a given shape needs to be stored in the computer's memory. This memory is just a long sequence of 0's and 1's (a whole bunch of on or off switches.) Each one of these switches is a bit, eight of them together is a byte. Imagine if we had eight bits (one byte) in sequence - how many ways can we configure these switches? The answer is (and doing a little research into binary numbers will prove this point) 256 possibilities, or a range of numbers between 0 and 255. We will use eight bit color for our grayscale range and 24 bit for full color (eight bits for each of the red, green, and blue color components). |

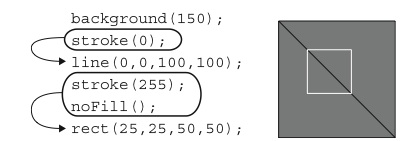
By adding the [**stroke()**](http://processing.org/reference/stroke_.html) and [**fill()**](http://processing.org/reference/fill_.html) functions before something is drawn, we can set the color of any given shape. There is also the function [**background()**](http://processing.org/reference/background.html), which sets a background color for the window. Here's an example.

background(255); // Setting the background to white

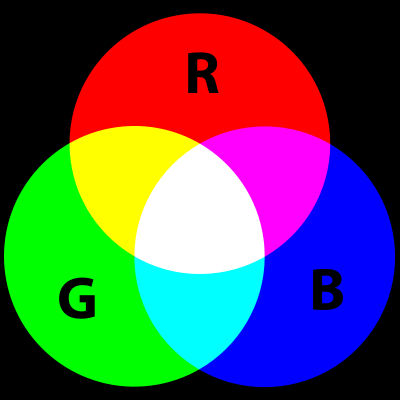
stroke(0); // Setting the outline (stroke) to black

fill(150); // Setting the interior of a shape (fill) to grey

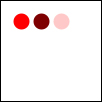
rect(50,50,75,100); // Drawing the rectangle

Stroke or fill can be eliminated with the functions: [noStroke()](http://processing.org/reference/noStroke_.html) and [noFill()](http://processing.org/reference/noFill_.html). Our instinct might be to say "stroke(0)" for no outline, however, it is important to remember that 0 is not "nothing", but rather denotes the color black. Also, remember not to eliminate both - with **noStroke()** and **noFill()**, nothing will appear!   
  
In addition, if we draw two shapes, Processing will always use the most recently specified stroke and fill, reading the code from top to bottom.   
  


## RGB Color

  
  
Remember finger painting? By mixing three "primary" colors, any color could be generated. Swirling all colors together resulted in a muddy brown. The more paint you added, the darker it got. Digital colors are also constructed by mixing three primary colors, but it works differently from paint. First, the primaries are diff erent: red, green, and blue (i.e., "RGB" color). And with color on the screen, you are mixing light, not paint, so the mixing rules are different as well.

* Red + Green = Yellow
* Red + Blue = Purple
* Green + Blue = Cyan (blue-green)
* Red + Green + Blue = White
* no colors = Black

This assumes that the colors are all as bright as possible, but of course, you have a range of color available, so some red plus some green plus some blue equals gray, and a bit of red plus a bit of blue equals dark purple. While this may take some getting used to, the more you program and experiment with RGB color, the more it will become instinctive, much like swirling colors with your fi ngers. And of course you can't say "Mix some red with a bit of blue," you have to provide an exact amount. As with grayscale, the individual color elements are expressed as ranges from 0 (none of that color) to 255 (as much as possible), and they are listed in the order R, G, and B. You will get the hang of RGB color mixing through experimentation, but next we will cover some code using some common colors.   
  
  
  
[Example: RGB color](http://www.learningprocessing.com/wp/examples/chapter-1/example-1-3/)

background(255);

noStroke();

// Bright red

fill(255,0,0);

ellipse(20,20,16,16);

// Dark red

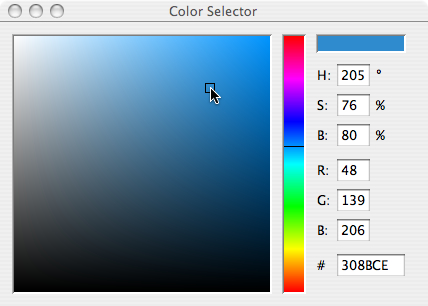
fill(127,0,0);

ellipse(40,20,16,16);

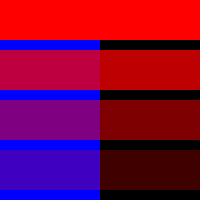
// Pink (pale red)

fill(255,200,200);

ellipse(60,20,16,16);

Processing also has a color selector to aid in choosing colors. Access this via TOOLS (from the menu bar) → COLOR SELECTOR.   
  


## Color Transparency

In addition to the red, green, and blue components of each color, there is an additional optional fourth component, referred to as the color's "alpha." Alpha means transparency and is particularly useful when you want to draw elements that appear partially see-through on top of one another. Th e alpha values for an image are sometimes referred to collectively as the "alpha channel" of an image.   
  
It is important to realize that pixels are not literally transparent, this is simply a convenient illusion that is accomplished by blending colors. Behind the scenes, Processing takes the color numbers and adds a percentage of one to a percentage of another, creating the optical perception of blending. (If you are interested in programming "rose-colored" glasses, this is where you would begin.)   
  
Alpha values also range from 0 to 255, with 0 being completely transparent (i.e., 0% opaque) and 255 completely opaque (i.e., 100% opaque).   
  
  
  
[Example: Alpha transparency](http://www.learningprocessing.com/wp/examples/chapter-1/example-1-4/)

size(200,200);

background(0);

noStroke();

// No fourth argument means 100% opacity.

fill(0,0,255);

rect(0,0,100,200);

// 255 means 100% opacity.

fill(255,0,0,255);

rect(0,0,200,40);

// 75% opacity.

fill(255,0,0,191);

rect(0,50,200,40);

// 55% opacity.

fill(255,0,0,127);

rect(0,100,200,40);

// 25% opacity.

fill(255,0,0,63);

rect(0,150,200,40);

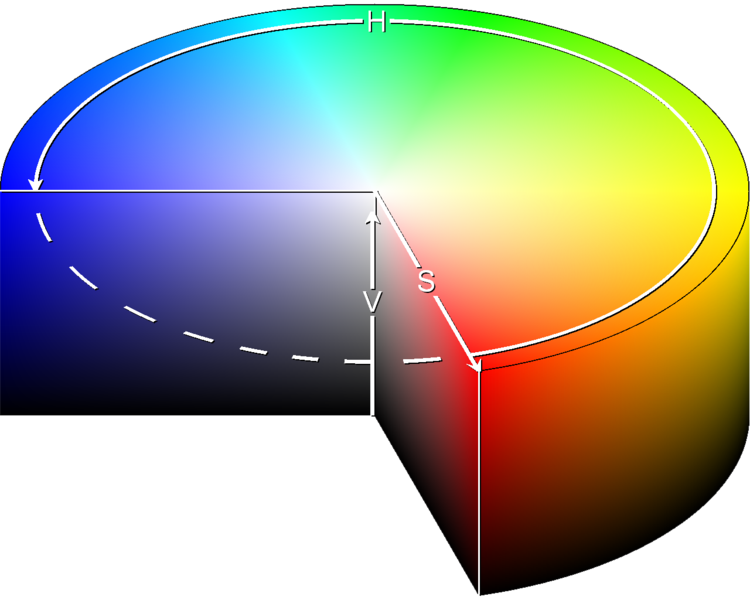
## Custom Color Ranges

RGB color with ranges of 0 to 255 is not the only way you can handle color in Processing. Behind the scenes in the computer's memory, color is always talked about as a series of 24 bits (or 32 in the case of colors with an alpha). However, Processing will let us think about color any way we like, and translate our values into numbers the computer understands. For example, you might prefer to think of color as ranging from 0 to 100 (like a percentage). You can do this by specifying a custom [**colorMode()**](http://processing.org/reference/colorMode_.html).

colorMode(RGB,100);

The above function says: "OK, we want to think about color in terms of red, green, and blue. The range of RGB values will be from 0 to 100."   
  
Although it is rarely convenient to do so, you can also have different ranges for each color component:

colorMode(RGB,100,500,10,255);

Now we are saying "Red values go from 0 to 100, green from 0 to 500, blue from 0 to 10, and alpha from 0 to 255."   
  
Finally, while you will likely only need RGB color for all of your programming needs, you can also specify colors in the HSB (hue, saturation, and brightness) mode. Without getting into too much detail, HSB color works as follows:   
  


* **Hue** - The color type, ranges from 0 to 360 by default (think of 360 degrees on a color "wheel").
* **Saturation** - The vibrancy of the color, 0 to 100 by default.
* **Brightness** - The, well, brightness of the color, 0 to 100 by default.

With [**colorMode()**](http://processing.org/reference/colorMode_.html) you can set your own color range.

# Drawing Curves

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *This tutorial is for Processing version 1.1+. If you see any errors or have comments, please* [*let us know*](http://code.google.com/p/processing/issues/list)*. Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*  This tutorial introduces you to the three types of curves in Processing: arcs, spline curves, and Bézier curves. Arcs Arcs are the simplest curves to draw. Processing defines an arc as a section of an ellipse. You call the function with these parameters:  arc(*x*, *y*, *width*, *height*, *start*, *stop*);  The first four parameters are the same as the ones for ellipse(); they define the boundary box for your arc. The last two parameters are the starting and ending angle for the arc. These angles, as with all other angles in Processing, are given in radians. Remember that angles are measured clockwise, with zero degrees pointing east. Using the fact that PI radians equals 180°, here are some example arcs.  Sample arcs  void setup()  {  size(300, 200);  background(255);  smooth();    rectMode(CENTER); // show bounding boxes  stroke(128);  rect(35, 35, 50, 50);  rect(105, 35, 50, 50);  rect(175, 35, 50, 50);  rect(105, 105, 100, 50);    stroke(0);  arc(35, 35, 50, 50, 0, PI / 2.0); // lower quarter circle  arc(105, 35, 50, 50, -PI, 0); // upper half of circle  arc(175, 35, 50, 50, -PI / 6, PI / 6); // 60 degrees  arc(105, 105, 100, 50, PI / 2, 3 \* PI / 2); // 180 degrees  } Spline Curves Arcs are fine, but they’re plain. The next function, curve(), lets you draw curves that aren’t necessarily part of an arc. This function draws what is technically called a *Rom-Catmull Spline*. To draw the curve, you have to specify the (*x*, *y*) coordinates of the points where the curve starts and ends. You must also specify two *control points* which determine the direction and amount of curvature. A call to curve() uses these parameters:  curve(*cpx1*, *cpy1*, *x1*, *y1*, *x2*, *y2*, *cpx2*, *cpy2*);   |  |  | | --- | --- | | cpx1, cpy1 | Coordinates of the first control point | | x1, y1 | Coordinates of the curve’s starting point | | x2, y2 | Coordinates of the curve’s ending point | | cpx2, cpy2 | Coordinates of the second control point |   Here is an example that shows a curve(). The control points are shown in red and the curve points in blue.  gentle curve with control points  void setup()  {  size(200, 200);  background(255);  smooth();  stroke(0);  curve(40, 40, 80, 60, 100, 100, 60, 120);    noStroke();  fill(255, 0, 0);  ellipse(40, 40, 3, 3);  fill(0, 0, 255, 192);  ellipse(100, 100, 3, 3);  ellipse(80, 60, 3, 3);  fill(255, 0, 0);  ellipse(60, 120, 3, 3);  }  curve showing relationship between control pointsHow do the control points affect the way the curve looks? Take a deep breath, because this is somewhat complicated.   * The tangent to the curve at the start point is parallel to the line between control point one and the end of the curve. These are the lines shown in green in the diagram at the left. * The tangent to the curve at the end point is parallel to the line between the start point and control point 2. These are the lines shown in purple in the diagram at the left.   The only way to understand this better is to play with it. [Here is an applet that lets you do just that.](http://www.processing.org/learning/curves/code/try_curve/index.html) Continuous Spline Curves In isolation, a single curve() is not particularly appealing. To draw a continuous curve through several points, you are better off using the curveVertex() function. You can only use this function when you are creating a shape with the beginShape() and endShape() functions.  Here is a curve connecting the points (40, 40), (80, 60), (100, 100), (60, 120), and (50, 150). In common usage, people use the first point of the curve as the first control point and the last point of the curve as the last control point. The lower diagram shows the tangents at the intermediate points.  curve showing vertices curve showing vertices and tangents  void setup()  {  int[ ] coords = {  40, 40, 80, 60, 100, 100, 60, 120, 50, 150  };  int i;    size(200, 200);  background(255);  smooth();  noFill();  stroke(0);  beginShape();  curveVertex(40, 40); // the first control point  curveVertex(40, 40); // is also the start point of curve  curveVertex(80, 60);  curveVertex(100, 100);  curveVertex(60, 120);  curveVertex(50, 150); // the last point of curve  curveVertex(50, 150); // is also the last control point  endShape();    // use the array to keep the code shorter;  // you already know how to draw ellipses!  fill(255, 0, 0);  noStroke();  for (i = 0; i < coords.length; i += 2)  {  ellipse(coords[i], coords[i + 1], 3, 3);  }    } Bézier Curves Though better than arcs, spline curves don’t seem to have those graceful, swooping curves that say “art.” For those, you need to draw Bézier curves with the bezier() function. As with spline curves, the bezier() function has six parameters, but, for some unknown reason, the order is different:  bezier(*x1*, *y1*, *cpx1*, *cpy1*, *cpx2*, *cpy2*, *x2*, *y2*);   |  |  | | --- | --- | | x1, y1 | Coordinates of the curve’s starting point | | cpx1, cpy1 | Coordinates of the first control point | | cpx2, cpy2 | Coordinates of the second control point | | x2, y2 | Coordinates of the curve’s ending point |   Here is a program that displays a Bézier curve and its control points.  Bézier curve with endpoints and control points marked  void setup( )  {  size(150, 150);  background(255);  smooth();  ellipse(50, 75, 5, 5); // endpoints of curve  ellipse(100, 75, 5, 5);  fill(255, 0, 0);  ellipse(25, 25, 5, 5); // control points  ellipse(125, 25, 5, 5);  noFill();  stroke(0);  bezier(50, 75, 25, 25, 125, 25, 100, 75);  }  Bézier curve showing control points connected to curve While it is difficult to visualize how the control points affect a curve(), it is slightly easier to see how the control points affect Bézier curves. Imagine two poles and several rubber bands. The poles connect the control points to the endpoints of the curve. A rubber band connects the tops of the poles. Two more rubber bands connect the midpoints of the poles to the midpoint of the first rubber band. One more rubber band connects *their* midpoints. The center of that last rubber band is tied to the curve.  Again, the only way to understand this better is to play with it. Continuous Bézier Curves Just as curveVertex() allows you to make continuous spline curves, bezierVertex() lets you make continuous Bézier curves. Again, you must be within a beginShape() / endShape() sequence. You must use vertex(*startX*, *startY*) to specify the starting point of the curve. Subsequent points are specified with a call to:  bezierVertex(*cpx1*, *cpy1*, *cpx2*, *cpy2*, *x*, *y*);   |  |  | | --- | --- | | cpx1, cpy1 | Coordinates of the first control point | | cpx2, cpy2 | Coordinates of the second control point | | x, y | The next point on the curve |   So, to draw the previous example using bezierVertex(), you would do this:  void setup( )  {  size(150, 150);  background(255);  smooth();  // don't show where control points are  noFill();  stroke(0);  beginShape();  vertex(50, 75); // first point  bezierVertex(25, 25, 125, 25, 100, 75);  endShape();  }  lumpy continuous bezier curveHere is a continuous Bézier curve, but it doesn’t join smoothly. The diagram shows the control points, but only the relevant code for drawing the curve is here.  beginShape();  vertex(30, 70); // first point  bezierVertex(25, 25, 100, 50, 50, 100);  bezierVertex(50, 140, 75, 140, 120, 120);  endShape();  smooth continuous bezier curve  In order to make two curves A and B smoothly continuous, the last control point of A, the last point of A, and the first control point of B have to be on a straight line. Here is an example that meets those conditions. The points that are in a line are shown in bold.  beginShape();  vertex(30, 70); // first point  bezierVertex(25, 25, **100, 50, 50, 100**);  bezierVertex(**20, 130**, 75, 140, 120, 120);  endShape(); Summary  * Use arc() when you need a segment of a circle or an ellipse. You can’t make continuous arcs or use them as part of a shape. * Use curve() when you need a small curve between two points. Use curveVertex() to make a continuous series of curves as part of a shape. * Use bezier() when you need long, smooth curves. Use bezierVertex() to make a continuous series of Bézier curves as part of a shape. |

# Strings and Drawing Text

This tutorial is for Processing version 1.1+. If you see any errors or have comments, please [*let us know*](http://code.google.com/p/processing/issues/list). This tutorials is from the book, [*Learning Processing*](http://www.processing.org/learning/books/#shiffman), by Daniel Shiffman, published by Morgan Kaufmann Publishers, Copyright 2008 Elsevier Inc. All rights reserved.  
*Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*

## The String class

If you are looking to display text onscreen with Processing, you've got to first become familiar with the [String](http://processing.org/reference/String.html) class. Strings are probably not a totally new concept for you, it's quite likely you've dealt with them before. For example, if you've printed some text to the message window or loaded an image from a file, you've written code like so:

println("printing some text to the message window!"); // Printing a String

PImage img = loadImage("filename.jpg"); // Using a String for a file name

Nevertheless, although you may have used a [String](http://processing.org/reference/String.html) here and there, it's time to unleash their full potential.

## Where do we find documentation for the String class?

Although technically a Java class, because Strings are so commonly used, Processing includes documentation in its reference: <http://www.processing.org/reference/String.html>.

This page only covers some of the available methods of the String class. The full documentation can be found on java's [String](http://download.oracle.com/javase/1.4.2/docs/api/java/lang/String.html) page.

## What is a String?

A String, at its core, is really just a fancy way of storing an array of characters. If we didn't have the String class, we'd probably have to write some code like this:

char[] sometext = {'H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd'};

Clearly, this would be a royal pain in the Processing behind. It's much simpler to do the following and make a String object:

String sometext = "How do I make String? Type some characters between quotation marks!";

It appears from the above that a String is nothing more than a list of characters in between quotes. Nevertheless, this is only the data of a String. We must remember that a String is an object with methods (which you can find on the reference page.) This is just like how we learned in the [Pixels tutorial](http://processing.org/learning/pixels/) that a [PImage](http://processing.org/reference/PImage.html) stores both the data associated with an image as well as the functionality: [copy()](http://processing.org/reference/copy_.html), [loadPixels()](http://processing.org/reference/loadPixels_.html), etc.

For example, the method [charAt()](http://processing.org/reference/String_charAt_.html) returns the individual character in the String at a given index. Note that Strings are just like arrays in that the first character is index #0!

String message = "some text here.";

char c = message.charAt(3);

println(c); // Results in 'e'

Another useful method is [length()](http://processing.org/reference/String_length_.html). This is easy to confuse with the length property of an array. However, when we ask for the length of a String object, we must use the parentheses since we are calling a function called length() rather than accessing a property called length.

String message = "This String is 34 characters long.";

println(message.length());

We can also change a String to all uppercase using the [toUpperCase()](http://processing.org/reference/String_toUpperCase_.html) method ([toLowerCase()](http://processing.org/reference/String_toLowerCase_.html) is also available).

String uppercase = message.toUpperCase();

println(uppercase);

You might notice something a bit odd here. Why didn't we simply say "message.toUpperCase()" and then print "message" variable? Instead, we assigned the result of "message.toUpperCase()" to a new variable with a different name -- "uppercase".

This is because a String is a special kind of object. It is immutable. An immutable object is one whose data can never be changed. Once we create a String, it stays the same for life. Anytime we want to change the String, we have to create a new one. So in the case of converting to uppercase, the method toUpperCase() returns a copy of the String object with all caps.

Finally, let's look at [equals()](http://processing.org/reference/String_equals_.html). Now, Strings can be compared with the "==" operator as follows:

String one = "hello";

String two = "hello";

println(one == two);

However, technically speaking, when "==" is used with objects, it compares the memory addresses for each object. Even though they contain the same data -- "hello"-- if they are different object instances "==" could result in a false comparison. The [equals()](http://processing.org/reference/String_equals_.html) function ensures that we are checking to see if two String objects contain the exact same sequence of characters, regardless of where that data is stored in the computer's memory.

String one = "hello";

String two = "hello";

println(one.equals(two));

Although both of the above methods return the correct result, it's safer to use equals(). Depending on how String objects are created in a sketch, "==" will not always work.

One other feature of String objects is concatenation, joining two Strings together. Strings are joined with the "+" operator. Plus, of course, usually means add in the case of numbers. When used with Strings, it means join.

String helloworld = "Hello" + "World";

Variables can also be brought into a String using concatenation.

int x = 10;

String message = "The value of x is: " + x;

## Displaying Text

The easiest way to display a String is to print it in the message window. This is likely something you've done while debugging. For example, if you needed to know the horizontal mouse location, you would write:

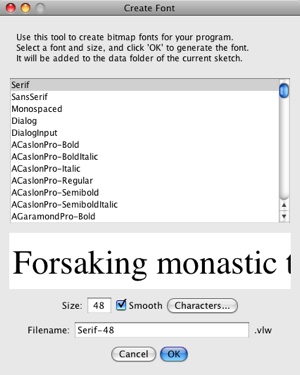
println(mouseX);

Or if you needed to determine that a certain part of the code was executed, you might print out a descriptive message.

println("We got here and we're printing out the mouse location!!!");

While this is valuable for debugging, it's not going to help our goal of displaying text for a user. To place text on screen, we have to follow a series of simple steps.

1. Choose a font by selecting "Tools" --> "Create Font." This will create and place the font file in your data directory. Make note of the font filename for Step 3. Processing uses a special font format, "vlw," that uses images to display each letter. Because of this, you should create the font at the size you intend to display.



2. Declare an object of type [PFont](http://processing.org/reference/PFont.html).

PFont f;

3. Load the font by referencing the font file name and the function [loadFont()](http://processing.org/reference/loadFont_.html). This should be done only once, usually in [setup()](http://processing.org/reference/setup_.html). Just as with loading an image, the process of loading a font into memory is slow and would seriously affect the sketch's performance if placed inside [draw()](http://processing.org/reference/draw_.html).

f = loadFont("Serif-48.vlw");

4. Specify the font using [textFont()](http://processing.org/reference/textFont_.html). textFont() takes one or two arguments, the font variable and the font size, which is optional. If you do not include the font size, the font will be displayed at the size originally loaded. Specifying a font size that is different from the font size loaded can also result in pixelated or poor quality text.

textFont(f,36);

5. Specify a color using [fill()](http://processing.org/reference/fill_.html).

fill(255);

6. Call the [text()](http://processing.org/reference/text_.html) function to display text. (This function is just like shape or image drawing, it takes 3 arguments -- the text to be displayed, and the x & y coordinate to display that text.)

text("Hello Strings!",10,100);

Here are all the steps together:

[**Example Simple Displaying Text**](http://www.learningprocessing.com/examples/chapter-17/example-17-1-simple-displaying-text/)

PFont f; // STEP 2 Declare PFont variable

void setup() {

size(200,200);

f = loadFont("ArialMT-16.vlw"); // STEP 3 Load Font

}

void draw() {

background(255);

textFont(f,16); // STEP 4 Specify font to be used

fill(0); // STEP 5 Specify font color

text("Hello Strings!",10,100); // STEP 6 Display Text

}

Fonts can also be created using the function [createFont()](http://processing.org/reference/createFont_.html).

f = createFont("Georgia", 24, true);

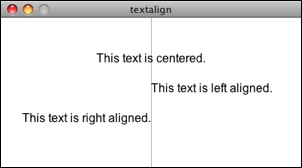
createFont() allows you to use a font that may be installed on your local machine, but not available as part of Processing's font options. In addition, createFont() allows the font to be scaled to any size without it looking pixelated or smushed. You can see all the available fonts with [PFont.list()](http://processing.org/reference/PFont_list_.html).

println(PFont.list());

## Animating Text

Let's look at two more useful Processing functions related to displaying text:

[textAlign()](http://processing.org/reference/textAlign_.html) -- specifies RIGHT, LEFT or CENTER alignment for text.



[**Example Text Align**](http://www.learningprocessing.com/examples/chapter-17/example-17-2/)

PFont f;

void setup() {

size(400,200);

f = createFont("Arial",16,true);

}

void draw() {

background(255);

stroke(175);

line(width/2,0,width/2,height);

textFont(f);

fill(0);

textAlign(CENTER);

text("This text is centered.",width/2,60);

textAlign(LEFT);

text("This text is left aligned.",width/2,100);

textAlign(RIGHT);

text("This text is right aligned.",width/2,140);

}

[textWidth()](http://processing.org/reference/textWidth_.html) -- Calculates and returns the width of any character or text string.

Let's say we want to create a news ticker, where text scrolls across the bottom of the screen from left to right. When the news headline leaves the window, it reappears on the right hand side and scrolls again. If we know the x location of the beginning of the text and we know the width of that text, we can determine when it is no longer in view. textWidth() gives us that width.

To start, we declare headline, font, and x location variables, initializing them in setup().

// A headline

String headline = "New study shows computer programming lowers cholesterol.";

PFont f; // Global font variable

float x; // horizontal location of headline

void setup() {

f = createFont("Arial",16,true); // Loading font

x = width; // initializing headline off-screen to the right

}

In draw(), we display the text at the appropriate location.

// Display headline at x location

textFont(f,16);

textAlign(LEFT);

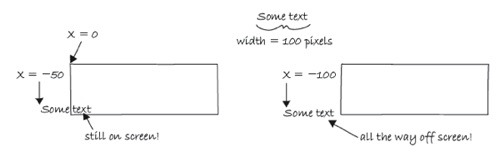
text(headline,x,180);

We change x by a speed value (in this case a negative number so that the text moves to the left.)

// Decrement x

x = x - 3;

Now comes more difficult part. It was easy to test when a circle reached the left side of the screen. We would simply ask: is x less than 0? With text, however, since it is left-aligned, when x equals zero, it is still viewable on screen. Instead, the text will be invisible when x is less than 0 minus the width of the text (See figure below). When that is the case, we reset x back to the right-hand side of the window, i.e. width.



// If x is less than the negative width, then it is completely off the screen

float w = textWidth(headline);

if (x < -w) {

x = width;

}

Here's the full example that displays a different headline each time the previous headline leaves the screen. The headlines are stored in a String array.

[**Example Scrolling Headlines**](http://www.learningprocessing.com/examples/chapter-17/example-17-3/)

// An array of news headlines

String[] headlines = {

"Processing downloads break downloading record.",

"New study shows computer programming lowers cholesterol.",

};

PFont f; // Global font variable

float x; // horizontal location of headline

int index = 0;

void setup() {

size(400,200);

f = createFont("Arial",16,true);

// Initialize headline offscreen to the right

x = width;

}

void draw() {

background(255);

fill(0);

// Display headline at x location

textFont(f,16);

textAlign(LEFT);

text(headlines[index],x,180);

// Decrement x

x = x - 3;

// If x is less than the negative width,

// then it is off the screen

float w = textWidth(headlines[index]);

if (x < -w) {

x = width;

index = (index + 1) % headlines.length;

}

}

In addition to textAlign() and textWidth(), Processing also offers the functions [textLeading()](http://processing.org/reference/textLeading_.html), [textMode()](http://processing.org/reference/textMode_.html), [textSize()](http://processing.org/reference/textSize_.html) for additional display functionality.

## Rotating text

[Translation and rotation](http://processing.org/learning/transform2d/) can also be applied to text. For example, to rotate text around its center, translate to an origin point and use textAlign(CENTER) before displaying the text.

[**Example: Rotating Text**](http://www.learningprocessing.com/examples/chapter-17/example-17-5-rotating-text/)

PFont f;

String message = "this text is spinning";

float theta;

void setup() {

size(200, 200);

f = createFont("Arial",20,true);

}

void draw() {

background(255);

fill(0);

textFont(f); // Set the font

translate(width/2,height/2); // Translate to the center

rotate(theta); // Rotate by theta

textAlign(CENTER);

text(message,0,0);

theta += 0.05; // Increase rotation

}

## Displaying text character by character

In certain graphics applications, displaying text with each character rendered individually is required. For example, if each character needs to move or be colored independently then simply saying

text("a bunch of letters",0,0);

will not do.

The solution is to loop through a String, displaying each character one at a time.

Let's start by looking at an example that displays the text all at once.

PFont f;

String message = "Each character is not written individually.";

void setup() {

size(400, 200);

f = createFont("Arial",20,true);

}

void draw() {

background(255);

fill(0);

textFont(f);

**// Displaying a block of text all at once using text().**

text(message,10,height/2);

}

We can rewrite the code to display each character in loop, using the [charAt()](http://processing.org/reference/String_charAt_.html) function.

String message = "Each character is written individually.";

// The first character is at pixel 10.

int x = 10;

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

// Each character is displayed one at a time with the charAt() function.

text(message.charAt(i),x,height/2);

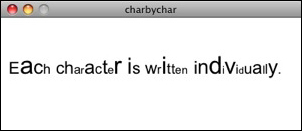
// All characters are spaced 10 pixels apart.

x += 10;

}

Calling the [text()](http://processing.org/reference/text_.html) function for each character will allow us more flexibility (for coloring, sizing, and placing characters within one String individually). The above code has a pretty major flaw, however -- the x location is increased by 10 pixels for each character. Although this is approximately correct, because each character is not exactly ten pixels wide, the spacing is off.

The proper spacing can be achieved using the [textWidth()](http://processing.org/reference/textWidth_.html) function as demonstrated in the code below. Note how this example achieves the proper spacing even with each character being a random size!



PFont f;

String message = "Each character is written individually.";

void setup() {

size(400, 150);

f = createFont("Arial",20,true);

}

void draw() {

background(255);

fill(0);

textFont(f);

int x = 10;

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

textSize(random(12,36));

text(message.charAt(i),x,height/2);

// textWidth() spaces the characters out properly.

x += textWidth(message.charAt(i));

}

noLoop();

}

This "letter by letter" methodology can also be applied to a sketch where characters from a String move independently of one another. The following example uses object-oriented design to make each character from the original String a Letter object, allowing it to both be a displayed in its proper location as well as move about the screen individually.

[**Example Text breaking up**](http://www.learningprocessing.com/examples/chapter-17/example-17-6/)

PFont f;

String message = "click mouse to shake it up";

// An array of Letter objects

Letter[] letters;

void setup() {

size(260, 200);

// Load the font

f = createFont("Arial",20,true);

textFont(f);

// Create the array the same size as the String

letters = new Letter[message.length()];

// Initialize Letters at the correct x location

int x = 16;

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

letters[i] = new Letter(x,100,message.charAt(i));

x += textWidth(message.charAt(i));

}

}

void draw() {

background(255);

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

// Display all letters

letters[i].display();

// If the mouse is pressed the letters shake

// If not, they return to their original location

if (mousePressed) {

letters[i].shake();

} else {

letters[i].home();

}

}

}

// A class to describe a single Letter

class Letter {

char letter;

// The object knows its original "home" location

float homex,homey;

// As well as its current location

float x,y;

Letter (float x\_, float y\_, char letter\_) {

homex = x = x\_;

homey = y = y\_;

letter = letter\_;

}

// Display the letter

void display() {

fill(0);

textAlign(LEFT);

text(letter,x,y);

}

// Move the letter randomly

void shake() {

x += random(-2,2);

y += random(-2,2);

}

// Return the letter home

void home() {

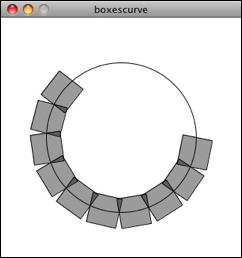
x = homex;

y = homey;

}

}

The character by character method also allows us to display text along a curve. Before we move on to letters, let's first look at how we would draw a series of boxes along a curve. This example makes heavy use of [Trignometry](http://www.processing.org/learning/trig/).



[**Example Boxes along a curve**](http://www.learningprocessing.com/examples/chapter-17/example-17-7/)

PFont f;

// The radius of a circle

float r = 100;

// The width and height of the boxes

float w = 40;

float h = 40;

void setup() {

size(320, 320);

smooth();

}

void draw() {

background(255);

// Start in the center and draw the circle

translate(width / 2, height / 2);

noFill();

stroke(0);

// Our curve is a circle with radius r in the center of the window.

ellipse(0, 0, r\*2, r\*2);

// 10 boxes along the curve

int totalBoxes = 10;

// We must keep track of our position along the curve

float arclength = 0;

// For every box

for (int i = 0; i < totalBoxes; i++) {

// Each box is centered so we move half the width

arclength += w/2;

// Angle in radians is the arclength divided by the radius

float theta = arclength / r;

pushMatrix();

// Polar to cartesian coordinate conversion

translate(r\*cos(theta), r\*sin(theta));

// Rotate the box

rotate(theta);

// Display the box

fill(0,100);

rectMode(CENTER);

rect(0,0,w,h);

popMatrix();

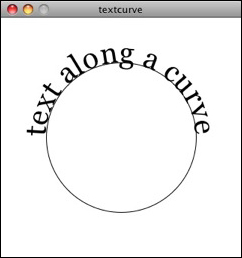
// Move halfway again

arclength += w/2;

}

}

What we need to do is replace each box with a character from a String that fits inside the box. And since characters all do not have the same width, instead of using a variable "w" that stays constant, each box will have a variable width along the curve according to the textWidth() function.



[**Example Characters along a curve**](http://www.learningprocessing.com/examples/chapter-17/example-17-8/)

// The message to be displayed

String message = "text along a curve";

PFont f;

// The radius of a circle

float r = 100;

void setup() {

size(320, 320);

f = createFont("Georgia",40,true);

textFont(f);

// The text must be centered!

textAlign(CENTER);

smooth();

}

void draw() {

background(255);

// Start in the center and draw the circle

translate(width / 2, height / 2);

noFill();

stroke(0);

ellipse(0, 0, r\*2, r\*2);

// We must keep track of our position along the curve

float arclength = 0;

// For every box

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

{

// Instead of a constant width, we check the width of each character.

char currentChar = message.charAt(i);

float w = textWidth(currentChar);

// Each box is centered so we move half the width

arclength += w/2;

// Angle in radians is the arclength divided by the radius

// Starting on the left side of the circle by adding PI

float theta = PI + arclength / r;

pushMatrix();

// Polar to cartesian coordinate conversion

translate(r\*cos(theta), r\*sin(theta));

// Rotate the box

rotate(theta+PI/2); // rotation is offset by 90 degrees

// Display the character

fill(0);

text(currentChar,0,0);

popMatrix();

// Move halfway again

arclength += w/2;

}

}

*Special thanks to* [*Ariel Malka*](http://ariel.chronotext.org/) *for his advice on this last curved text example.*

# Images and Pixels

*This tutorial is for Processing version 1.1+. If you see any errors or have comments, please* [*let us know*](http://code.google.com/p/processing/issues/list)*. This tutorials is from the book,* [*Learning Processing*](http://www.processing.org/learning/books/#shiffman)*, by Daniel Shiffman, published by Morgan Kaufmann Publishers, Copyright 2008 Elsevier Inc. All rights reserved.*  
*Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*

A digital image is nothing more than data -- numbers indicating variations of red, green, and blue at a particular location on a grid of pixels. Most of the time, we view these pixels as miniature rectangles sandwiched together on a computer screen. With a little creative thinking and some lower level manipulation of pixels with code, however, we can display that information in a myriad of ways. This tutorial is dedicated to breaking out of simple shape drawing in Processing and using images (and their pixels) as the building blocks of Processing graphics.

## Getting started with images

Hopefully, you are comfortable with the idea of data types. You probably specify them often -- a float variable "speed", an int "x", etc. These are all primitive data types, bits sitting in the computer's memory ready for our use. Though perhaps a bit trickier, you hopefully also use objects, complex data types that store multiple pieces of data (along with functionality) -- a "Ball" class, for example, might include floating point variables for location, size, and speed as well as methods to move, display itself, and so on.   
  
In addition to user-defined objects (such as Ball), Processing has a bunch of handy classes all ready to go without us writing any code. In this tutorial, we'll examine [PImage](http://processing.org/reference/PImage.html), a class for loading and displaying an image as well as looking at its pixels.   
  
[Example: "Hello World" images](http://www.learningprocessing.com/examples/chapter-15/example-15-1/)

// Declaring a variable of type PImage

PImage img;

void setup() {

size(320,240);

// Make a new instance of a PImage by loading an image file

img = loadImage("mysummervacation.jpg");

}

void draw() {

background(0);

// Draw the image to the screen at coordinate (0,0)

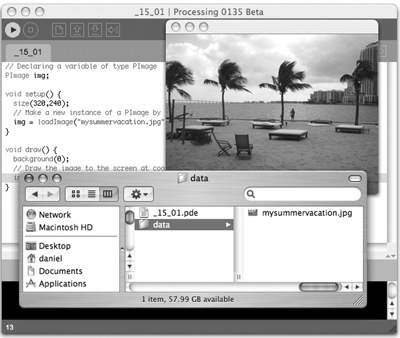
image(img,0,0);

}

Using an instance of a [PImage](http://processing.org/reference/PImage.html) object is no different than using a user-defined class. First, a variable of type [PImage](http://processing.org/reference/PImage.html), named "img," is declared. Second, a new instance of a [PImage](http://processing.org/reference/PImage.html) object is created via the [loadImage()](http://processing.org/reference/loadImage_.html) method. [loadImage()](http://processing.org/reference/loadImage_.html) takes one argument, a String indicating a file name, and loads the that file into memory. [loadImage()](http://processing.org/reference/loadImage_.html) looks for image files stored in your Processing sketch's "data" folder.

|  |
| --- |
|  |

|  |
| --- |
| **The Data Folder: How do I get there?**   Images can be added to the data folder automatically via:   Sketch --> Add File. . .   or manually:   Sketch --> Show Sketch Folder   This will open up the sketch folder. If there is no data directory create one. Otherwise, place your image files inside.   Processing accepts the following file formats for images: GIF, JPG, TGA, PNG. |

  
  
  
In the above example, it may seem a bit peculiar that we never called a "constructor" to instantiate the [PImage](http://processing.org/reference/PImage.html) object, saying "new PImage()". After all, in most object-related examples, a constructor is a must for producing an object instance.

Spaceship ss = new Spaceship();

Flower flr = new Flower(25);

yet:

PImage img = loadImage("file.jpg");

In fact, the [loadImage()](http://processing.org/reference/loadImage_.html) function performs the work of a constructor, returning a brand new instance of a [PImage](http://processing.org/reference/PImage.html) object generated from the specified filename. We can think of it as the [PImage](http://processing.org/reference/PImage.html) constructor for loading images from a file. For creating a blank image, the create[image()](http://processing.org/reference/image_.html) function is used.

// Create a blank image, 200x200 pixels with RGB color

PImage img = createImage(200,200,RGB);

We should also note that the process of loading the image from the hard drive into memory is a slow one, and we should make sure our program only has to do it once, in setup(). Loading images in draw() may result in slow performance as well as "Out of Memory" errors.   
  
Once the image is loaded, it is displayed with the [image()](http://processing.org/reference/image_.html) function. The [image()](http://processing.org/reference/image_.html) function must include 3 arguments -- the image to be displayed, the x location, and the y location. Optionally two arguments can be added to resize the image to a certain width and height.

image(img,10,20,90,60);

## Your very first image processing filter

When displaying an image, you might like to alter its appearance. Perhaps you would like the image to appear darker, transparent, blue-ish, etc. This type of simple image filtering is achieved with Processing's tint() function. tint() is essentially the image equivalent of shape's fill(), setting the color and alpha transparency for displaying an image on screen. An image, nevertheless, is not usually all one color. The arguments for tint() simply specify how much of a given color to use for every pixel of that image, as well as how transparent those pixels should appear.   
  
For the following examples, we will assume that two images (a sunflower and a dog) have been loaded and the dog is displayed as the background (which will allow us demonstrate transparency.)

PImage sunflower = loadImage("sunflower.jpg");

PImage dog = loadImage("dog.jpg");

background(dog);

If tint() receives one argument, only the brightness of the image is affected.   
  


// The image retains its original state.

tint(255);

image(sunflower,0,0);



//The image appears darker.

tint(100);

image(sunflower,0,0);

A second argument will change the image's alpha transparency.   
  


// The image is at 50% opacity.

tint(255,127);

image(sunflower,0,0);

Three arguments affect the brightness of the red, green, and blue components of each color.   
  


// None of its red, most of its green, and all of its blue.

tint(0,200,255);

image(sunflower,0,0);

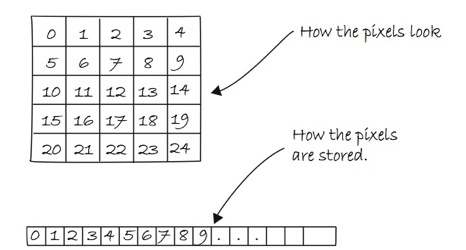
Finally, adding a fourth argument to the method manipulates the alpha (same as with 2). Incidentally, the range of values for tint() can be specified with colorMode().   
  


// The image is tinted red and transparent.

tint(255,0,0,100);

image(sunflower,0,0);

## Pixels, pixels, and more pixels

If you've just begun using Processing you may have mistakenly thought that the only offered means for drawing to the screen is through a function call. "Draw a line between these points" or "Fill an ellipse with red" or "load this JPG image and place it on the screen here." But somewhere, somehow, someone had to write code that translates these function calls into setting the individual pixels on the screen to reflect the requested shape. A line doesn't appear because we say [line()](http://processing.org/reference/line_.html), it appears because we color all the pixels along a linear path between two points. Fortunately, we don't have to manage this lower-level-pixel-setting on a day-to-day basis. We have the developers of Processing (and Java) to thank for the many drawing functions that take care of this business.   
  
Nevertheless, from time to time, we do want to break out of our mundane shape drawing existence and deal with the pixels on the screen directly. Processing provides this functionality via the pixels array.   
  
We are familiar with the idea of each pixel on the screen having an X and Y position in a two dimensional window. However, the array pixels has only one dimension, storing color values in linear sequence.   
  
  
  
Take the following simple example. This program sets each pixel in a window to a random grayscale value. The pixels array is just like an other array, the only difference is that we don't have to declare it since it is a Processing built-in variable.   
  
[Example: Setting Pixels](http://www.learningprocessing.com/examples/chapter-15/example-15-5-setting-pixels/)

size(200, 200);

// Before we deal with pixels

loadPixels();

// Loop through every pixel

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

// Pick a random number, 0 to 255

float rand = random(255);

// Create a grayscale color based on random number

color c = color(rand);

// Set pixel at that location to random color

pixels[i] = c;

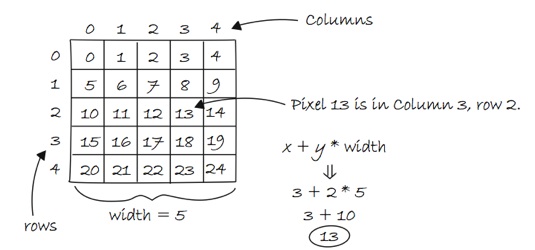
}

// When we are finished dealing with pixels

updatePixels();

First, we should point out something important in the above example. Whenever you are accessing the pixels of a Processing window, you must alert Processing to this activity. This is accomplished with two functions:

* [loadPixels()](http://processing.org/reference/loadPixels_.html) This function is called before you access the pixel array, saying "load the pixels, I would like to speak with them!"
* [updatePixels()](http://processing.org/reference/updatePixels.html) This function is called after you finish with the pixel array saying "Go ahead and update the pixels, I'm all done!"

In the above example, because the colors are set randomly, we didn't have to worry about where the pixels are onscreen as we access them, since we are simply setting all the pixels with no regard to their relative location. However, in many image processing applications, the XY location of the pixels themselves is crucial information. A simple example of this might be, set every even column of pixels to white and every odd to black. How could you do this with a one dimensional pixel array? How do you know what column or row any given pixel is in?   
  
In programming with pixels, we need to be able to think of every pixel as living in a two dimensional world, but continue to access the data in one (since that is how it is made available to us). We can do this via the following formula:   
  
1. Assume a window or image with a given WIDTH and HEIGHT.  
2. We then know the pixel array has a total number of elements equaling WIDTH \* HEIGHT.  
3. For any given X, Y point in the window, the location in our 1 dimensional pixel array is:   
  
**LOCATION = X + Y\*WIDTH**   
  
  
  
This may remind you of our [two dimensional arrays tutorial](http://www.processing.org/learning/2darray/). In fact, we'll need to use the same nested for loop technique. The difference is that, although we want use for loops to think about the pixels in two dimensions, when we go to actually access the pixels, they live in a one dimensional array, and we have to apply the formula from the above illustration.   
  
Let's look at how it is done.   
  
[Example: Setting Pixels according to their 2D location](http://www.learningprocessing.com/examples/chapter-15/example-15-6/)

size(200, 200);

loadPixels();

// Loop through every pixel column

for (int x = 0; x < width; x++) {

// Loop through every pixel row

for (int y = 0; y < height; y++) {

// Use the formula to find the 1D location

int loc = x + y \* width;

if (x % 2 == 0) { // If we are an even column

pixels[loc] = color(255);

} else { // If we are an odd column

pixels[loc] = color(0);

}

}

}

updatePixels();

## Intro To Image Processing

The previous section looked at examples that set pixel values according to an arbitrary calculation. We will now look at how we might set pixels according those found in an existing [PImage](http://processing.org/reference/PImage.html) object. Here is some pseudo-code.   
  
(1) Load the image file into a PImage object  
(2) For each pixel in the PImage, retrieve the pixel's color and set the display pixel to that color.   
  
The [PImage](http://processing.org/reference/PImage.html) class includes some useful fields that store data related to the image -- width, height, and pixels. Just as with our user-defined classes, we can access these fields via the dot syntax.

PImage img = createImage(320,240,RGB); // Make a PImage object

println(img.width); // Yields 320

println(img.height); // Yields 240

img.pixels[0] = color(255,0,0); // Sets the first pixel of the image to red

Access to these fields allows us to loop through all the pixels of an image and display them onscreen.   
  
[Example: Displaying the pixels of an image](http://www.learningprocessing.com/examples/chapter-15/example-15-7/)

PImage img;

void setup() {

size(200, 200);

img = loadImage("sunflower.jpg");

}

void draw() {

loadPixels();

// Since we are going to access the image's pixels too

img.loadPixels();

for (int y = 0; y < height; y++) {

for (int x = 0; x < width; x++) {

int loc = x + y\*width;

// The functions red(), green(), and blue() pull out the 3 color components from a pixel.

float r = red(img.pixels[loc]);

float g = green(img.pixels[loc]);

float b = blue(img.pixedls[loc];

// Image Processing would go here

// If we were to change the RGB values, we would do it here, before setting the pixel in the display window.

// Set the display pixel to the image pixel

pixels[loc] = color(r,g,b);

}

}

updatePixels();

}

Now, we could certainly come up with simplifications in order to merely display the image (for example, the nested loop is not required, not to mention that using the [image()](http://processing.org/reference/image_.html) function would allow us to skip all this pixel work entirely.) However, example 15-7 provides a basic framework for getting the red, green, and blue values for each pixel based on its spatial orientation (XY location); ultimately, this will allow us to develop more advanced image processing algorithms.   
  
Before we move on, I should stress that this example works because the display area has the same dimensions as the source image. If this were not the case, you would simply have to have two pixel location calculations, one for the source image and one for the display area.

int imageLoc = x + y\*img.width;

int displayLoc = x + y\*width;

## Our second image filter, making our own "tint"

Just a few paragraphs ago, we were enjoying a relaxing coding session, colorizing images and adding alpha transparency with the friendly tint() method. For basic filtering, this method did the trick. The pixel by pixel method, however, will allow us to develop custom algorithms for mathematically altering the colors of an image. Consider brightness -- brighter colors have higher values for their red, green, and blue components. It follows naturally that we can alter the brightness of an image by increasing or decreasing the color components of each pixel. In the next example, we dynamically increase or decrease those values based on the mouse's horizontal location. (Note, the next two examples include only the image processing loop itself, the rest of the code is assumed.)   
  
[Example: Adjusting image brightness](http://www.learningprocessing.com/examples/chapter-15/example-15-8/)

for (int x = 0; x < img.width; x++) {

for (int y = 0; y < img.height; y++ ) {

// Calculate the 1D pixel location

int loc = x + y\*img.width;

// Get the R,G,B values from image

float r = red (img.pixels[loc]);

float g = green (img.pixels[loc]);

float b = blue (img.pixels[loc]);

// Change brightness according to the mouse here

float adjustBrightness = ((float) mouseX / width) \* 8.0;

r \*= adjustBrightness;

g \*= adjustBrightness;

b \*= adjustBrightness;

// Constrain RGB to between 0-255

r = constrain(r,0,255);

g = constrain(g,0,255);

b = constrain(b,0,255);

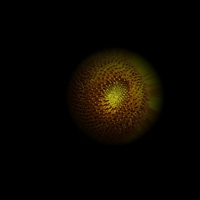
// Make a new color and set pixel in the window

color c = color(r,g,b);

pixels[loc] = c;

}

}

Since we are altering the image on a per pixel basis, all pixels need not be treated equally. For example, we can alter the brightness of each pixel according to its distance from the mouse.   
  
  
  
[Example: Adjusting image brightness based on pixel location](http://www.learningprocessing.com/examples/chapter-15/example-15-9/)

for (int x = 0; x < img.width; x++) {

for (int y = 0; y < img.height; y++ ) {

// Calculate the 1D pixel location

int loc = x + y\*img.width;

// Get the R,G,B values from image

float r = red (img.pixels[loc]);

float g = green (img.pixels[loc]);

float b = blue (img.pixels[loc]);

// Calculate an amount to change brightness

// based on proximity to the mouse

float distance = dist(x,y,mouseX,mouseY);

float adjustBrightness = (50-distance)/50;

r \*= adjustBrightness;

g \*= adjustBrightness;

b \*= adjustBrightness;

// Constrain RGB to between 0-255

r = constrain(r,0,255);

g = constrain(g,0,255);

b = constrain(b,0,255);

// Make a new color and set pixel in the window

color c = color(r,g,b);

pixels[loc] = c;

}

}

## Writing to another PImage object's pixels

All of our image processing examples have read every pixel from a source image and written a new pixel to the Processing window directly. However, it's often more convenient to write the new pixels to a destination image (that you then display using the [image()](http://processing.org/reference/image_.html) function). We'll demonstrate this technique while looking at another simple pixel operation: threshold.   
  
A threshold filter displays each pixel of an image in only one of two states, black or white. That state is set according to a particular threshold value. If the pixel's brightness is greater than the threshold, we color the pixel white, less than, black. In the code below, we use an arbitrary threshold of 100.   
  
  
  
[Example: Brightness Threshold](http://www.learningprocessing.com/examples/chapter-15/example-15-10/)

PImage source; // Source image

PImage destination; // Destination image

void setup() {

size(200, 200);

source = loadImage("sunflower.jpg");

// The destination image is created as a blank image the same size as the source.

destination = createImage(source.width, source.height, RGB);

}

void draw() {

float threshold = 127;

// We are going to look at both image's pixels

source.loadPixels();

destination.loadPixels();

for (int x = 0; x < source.width; x++) {

for (int y = 0; y < source.height; y++ ) {

int loc = x + y\*source.width;

// Test the brightness against the threshold

if (brightness(source.pixels[loc]) > threshold) {

destination.pixels[loc] = color(255); // White

} else {

destination.pixels[loc] = color(0); // Black

}

}

}

// We changed the pixels in destination

destination.updatePixels();

// Display the destination

image(destination,0,0);

}

This particular functionality is available without per pixel processing as part of Processing's filter() function. Understanding the lower level code, however, is crucial if you want to implement your own image processing algorithms, not available with filter().   
  
But if all you want to do is threshold, here is how:

// Draw the image

image(img,0,0);

// Filter the window with a threshold effect

// 0.5 means threshold is 50% brightness

filter(THRESHOLD,0.5);

## Level II: Pixel Group Processing

In previous examples, we've seen a one-to-one relationship between source pixels and destination pixels. To increase an image's brightness, we take one pixel from the source image, increase the RGB values, and display one pixel in the output window. In order to perform more advanced image processing functions, we must move beyond the one-to-one pixel paradigm into pixel group processing.   
  
Let's start by creating a new pixel out of a two pixels from a source image -- a pixel and its neighbor to the left.   
  
If we know the pixel is located at (x,y):

int loc = x + y\*img.width;

color pix = img.pixels[loc];

Then its left neighbor is located at (x-1,y):

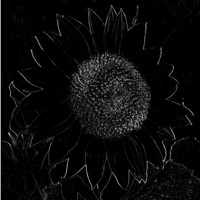
int leftLoc = (x-1) + y\*img.width;

color leftPix = img.pixels[leftLoc];

We could then make a new color out of the difference between the pixel and its neighbor to the left.

float diff = abs(brightness(pix) - brightness(leftPix));

pixels[loc] = color(diff);

Here is the full algorithm:   
  
  
  
[Example: Pixel neighbor differences (edges)](http://www.learningprocessing.com/examples/chapter-15/example-15-12/)

// Since we are looking at left neighbors

// We skip the first column

for (int x = 1; x < width; x++) {

for (int y = 0; y < height; y++ ) {

// Pixel location and color

int loc = x + y\*img.width;

color pix = img.pixels[loc];

// Pixel to the left location and color

int leftLoc = (x-1) + y\*img.width;

color leftPix = img.pixels[leftLoc];

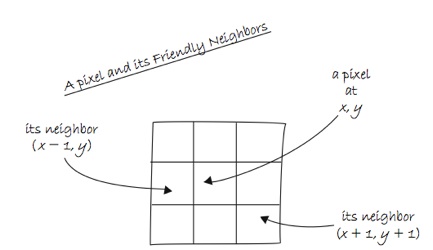
// New color is difference between pixel and left neighbor

float diff = abs(brightness(pix) - brightness(leftPix));

pixels[loc] = color(diff);

}

}

This example is a simple horizontal edge detection algorithm. When pixels differ greatly from their neighbors, they are most likely "edge" pixels. For example, think of a picture of white piece of paper on a black tabletop. The edges of that paper are where the colors are most different, where white meets black.   
  
In the previous example, we looked at two pixels to find edges. More sophisticated algorithms, however, usually involve looking at many pixels at a time. After all, each pixel has 8 immediate neighbors: top left, top, top right, right, bottom right, bottom, bottom left, left.   
  
  
  
These image processing algorithms are often referred to as a "spatial convolution." The process uses a weighted average of an input pixel and its neighbors to calculate an output pixel. In other words, that new pixel is a function of an area of pixels. Neighboring areas of different sizes can be employed, such as a 3x3 matrix, 5x5, etc.   
  
Different combinations of weights for each pixel result in various effects. For example, we "sharpen" an image by subtracting the neighboring pixel values and increasing the center point pixel. A blur is achieved by taking the average of all neighboring pixels. (Note that the values in the convolution matrix add up to 1).   
  
For example,

Sharpen:

-1 -1 -1

-1 9 -1

-1 -1 -1

Blur:

1/9 1/9 1/9

1/9 1/9 1/9

1/9 1/9 1/9

Following is an example that performs a convolution using a 2D array (see Chapter 13, p. XX for a review of 2D arrays) to store the pixel weights of a 3x3 matrix. This example is probably the most advanced example we've encountered in this book so far since it involves so many elements (nested loops, 2D arrays, PImage pixels, and so on.)   
  
  
  
[Example: Sharpen with Convolution](http://www.learningprocessing.com/examples/chapter-15/example-15-13/)

PImage img;

int w = 80;

// It's possible to perform a convolution

// the image with different matrices

float[][] matrix = { { -1, -1, -1 },

{ -1, 9, -1 },

{ -1, -1, -1 } };

void setup() {

size(200, 200);

frameRate(30);

img = loadImage("sunflower.jpg");

}

void draw() {

// We're only going to process a portion of the image

// so let's set the whole image as the background first

image(img,0,0);

// Where is the small rectangle we will process

int xstart = constrain(mouseX-w/2,0,img.width);

int ystart = constrain(mouseY-w/2,0,img.height);

int xend = constrain(mouseX+w/2,0,img.width);

int yend = constrain(mouseY+w/2,0,img.height);

int matrixsize = 3;

loadPixels();

// Begin our loop for every pixel

for (int x = xstart; x < xend; x++) {

for (int y = ystart; y < yend; y++ ) {

// Each pixel location (x,y) gets passed into a function called convolution()

// which returns a new color value to be displayed.

color c = convolution(x,y,matrix,matrixsize,img);

int loc = x + y\*img.width;

pixels[loc] = c;

}

}

updatePixels();

stroke(0);

noFill();

rect(xstart,ystart,w,w);

}

color convolution(int x, int y, float[][] matrix, int matrixsize, PImage img) {

float rtotal = 0.0;

float gtotal = 0.0;

float btotal = 0.0;

int offset = matrixsize / 2;

// Loop through convolution matrix

for (int i = 0; i < matrixsize; i++){

for (int j= 0; j < matrixsize; j++){

// What pixel are we testing

int xloc = x+i-offset;

int yloc = y+j-offset;

int loc = xloc + img.width\*yloc;

// Make sure we have not walked off the edge of the pixel array

loc = constrain(loc,0,img.pixels.length-1);

// Calculate the convolution

// We sum all the neighboring pixels multiplied by the values in the convolution matrix.

rtotal += (red(img.pixels[loc]) \* matrix[i][j]);

gtotal += (green(img.pixels[loc]) \* matrix[i][j]);

btotal += (blue(img.pixels[loc]) \* matrix[i][j]);

}

}

// Make sure RGB is within range

rtotal = constrain(rtotal,0,255);

gtotal = constrain(gtotal,0,255);

btotal = constrain(btotal,0,255);

// Return the resulting color

return color(rtotal,gtotal,btotal);

}

## Visualizing the Image

You may be thinking: "Gosh, this is all very interesting, but seriously, when I want to blur an image or change its brightness, do I really need to write code? I mean, can't I use Photoshop?" Indeed, what we have achieved here is an merely an introductory understanding of what highly skilled programmers at Adobe do. The power of Processing, however, is the potential for real-time, interactive graphics applications. There is no need for us to live within the confines of "pixel point" and "pixel group" processing.   
  
Following are two examples of algorithms for drawing processing shapes. Instead of coloring the shapes randomly or with hard-coded values as we have in the past, we select colors from pixels inside of a [PImage](http://processing.org/reference/PImage.html) object. The image itself is never displayed; rather, it serves as a database of information that we can exploit for a multitude of creative pursuits.   
  
In this first example, for every cycle through draw(), we fill one ellipse at a random location onscreen with a color taken from its corresponding location in the source image. The result is a basic "pointillist-like" effect:   
  
  
  
[Example: "Pointillism"](http://www.learningprocessing.com/examples/chapter-15/example-15-14/)

PImage img;

int pointillize = 16;

void setup() {

size(200,200);

img = loadImage("sunflower.jpg");

background(0);

smooth();

}

void draw() {

// Pick a random point

int x = int(random(img.width));

int y = int(random(img.height));

int loc = x + y\*img.width;

// Look up the RGB color in the source image

loadPixels();

float r = red(img.pixels[loc]);

float g = green(img.pixels[loc]);

float b = blue(img.pixels[loc]);

noStroke();

// Draw an ellipse at that location with that color

fill(r,g,b,100);

ellipse(x,y,pointillize,pointillize);

}

In this next example, we take the data from a two-dimensional image and using the 3D translation techniques described in chapter 14, render a rectangle for each pixel in three-dimensional space. The z location is determined by the brightness of the color. Brighter colors appear closer to the viewer and darker ones farther away.   
  
  
  
[Example: 2D image mapped to 3D](http://www.learningprocessing.com/examples/chapter-15/example-15-15/)

PImage img; // The source image

int cellsize = 2; // Dimensions of each cell in the grid

int cols, rows; // Number of columns and rows in our system

void setup() {

size(200, 200, P3D);

img = loadImage("sunflower.jpg"); // Load the image

cols = width/cellsize; // Calculate # of columns

rows = height/cellsize; // Calculate # of rows

}

void draw() {

background(0);

loadPixels();

// Begin loop for columns

for ( int i = 0; i < cols;i++) {

// Begin loop for rows

for ( int j = 0; j < rows;j++) {

int x = i\*cellsize + cellsize/2; // x position

int y = j\*cellsize + cellsize/2; // y position

int loc = x + y\*width; // Pixel array location

color c = img.pixels[loc]; // Grab the color

// Calculate a z position as a function of mouseX and pixel brightness

float z = (mouseX/(float)width) \* brightness(img.pixels[loc]) - 100.0;

// Translate to the location, set fill and stroke, and draw the rect

pushMatrix();

translate(x,y,z);

fill(c);

noStroke();

rectMode(CENTER);

rect(0,0,cellsize,cellsize);

popMatrix();

}

}

}

# 2-D Transformations

|  |
| --- |
| *This tutorial is for Processing version 1.1+. If you see any errors or have comments, please* [*let us know*](http://code.google.com/p/processing/issues/list)*.* *Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*  Processing has built-in functions that make it easy for you to have objects in a sketch move, spin, and grow or shrink. This tutorial will introduce you to the translate, rotate, and scale functions so that you can use them in your sketches. Translation: Moving the Grid As you know, your Processing window works like a piece of graph paper. When you want to draw something, you specify its coordinates on the graph. Here is a simple rectangle drawn with the code rect(20, 20, 40, 40). The coordinate system (a fancy word for “graph paper”) is shown in gray.  Black rectangle on gray numbered grid  If you want to move the rectangle 60 units right and 80 units down, you can just change the coordinates by adding to the *x* and *y* starting point: rect(20 + 60, 20 + 80, 40, 40) and the rectangle will appear in a different place. (We put the arrow in there for dramatic effect.)  Black rectangle on gray numbered grid, moved  But there is a more interesting way to do it: **move the graph paper instead**. If you move the graph paper 60 units right and 80 units down, you will get exactly the same visual result. Moving the coordinate system is called *translation*.  grid moved with arrow showing motion  The important thing to notice in the preceding diagram is that, as far as the rectangle is concerned, it hasn’t moved at all. Its upper left corner is still at (20,20). When you use transformations, the things you draw *never* change position; the coordinate system does.  Here is code that draws the rectangle in red by changing its coordinates, then draws it in blue by moving the grid. The rectangles are translucent so that you can see that they are (visually) at the same place. Only the method used to move them has changed. Copy and paste this code into Processing and give it a try.  void setup()  {  size(200, 200);  background(255);  noStroke();  // draw the original position in gray  fill(192);  rect(20, 20, 40, 40);    // draw a translucent red rectangle by changing the coordinates  fill(255, 0, 0, 128);  rect(20 + 60, 20 + 80, 40, 40);    // draw a translucent blue rectangle by translating the grid  fill(0, 0, 255, 128);  pushMatrix();  translate(60, 80);  rect(20, 20, 40, 40);  popMatrix();  }  Let’s look at the translation code in more detail. pushMatrix() is a built-in function that saves the current position of the coordinate system. The translate(60, 80) moves the coordinate system 60 units right and 80 units down. The rect(20, 20, 40, 40) draws the rectangle at the same place it was originally. Remember, the things you draw don’t move—the grid moves instead. Finally, popMatrix() restores the coordinate system to the way it was before you did the translate.  Yes, you could have done a translate(-60, -80) to move the grid back to its original position. However, when you start doing more sophisticated operations with the coordinate system, it’s easier to use pushMatrix() and popMatrix() to save and restore the status rather than having to undo all your operations. Later on in this tutorial, you will find out why those functions seem to have such strange names. What’s the Advantage? You may be thinking that picking up the coordinate system and moving it is a lot more trouble than just adding to coordinates. For a simple example like the rectangle, you are correct. But let’s take an example of where translate() can make life easier. Here is some code that draws a row of houses. It uses a loop that calls function named house(), which takes the *x* and *y* location of the house’s upper-left corner as its parameters.  Row of stick-figure houses  void setup()  {  size(400, 100);  background(255);  for (int i = 10; i < 350; i = i + 50)  {  house(i, 20);  }  }  This is the code for drawing the house by changing its position. Look at all the additions that you have to keep track of.  void house(int x, int y)  {  triangle(x + 15, y, x, y + 15, x + 30, y + 15);  rect(x, y + 15, 30, 30);  rect(x + 12, y + 30, 10, 15);  }  Compare that to the version of the function that uses translate(). In this case, the code draws the house in the same place every time, with its upper left corner at (0, 0), and lets translation do all the work instead.  void house(int x, int y)  {  pushMatrix();  translate(x, y);  triangle(15, 0, 0, 15, 30, 15);  rect(0, 15, 30, 30);  rect(12, 30, 10, 15);  popMatrix();  } Rotation In addition to moving the grid, you can also rotate it with the rotate() function. This function takes one argument, which is the number of *radians* that you want to rotate. In Processing, all the functions that have to do with rotation measure angles in radians rather than degrees. When you talk about angles in degrees, you say that a full circle has 360°. When you talk about angles in radians, you say that a full circle has 2π radians. Here is a diagram of how Processing measures angles in degrees (black) and radians (red).  Degrees are measured clockwise with zero being at 3 o'clock  Since most people think in degrees, Processing has a built-in radians() function which takes a number of degrees as its argument and converts it for you. It also has a degrees() function that converts radians to degrees. Given that background, let’s try rotating a square clockwise 45 degrees.  square has moved and rotated  void setup()  {  size(200, 200);  background(255);  smooth();  fill(192);  noStroke();  rect(40, 40, 40, 40);    pushMatrix();  rotate(radians(45));  fill(0);  rect(40, 40, 40, 40);  popMatrix();  }  Hey, what happened? How come the square got moved and cut off? The answer is: the square did not move. The **grid** was rotated. Here is what really happened. As you can see, on the rotated coordinate system, the square still has its upper left corner at (40, 40).  shows grid rotated 45 degrees clockwise Rotating the Correct Way The correct way to rotate the square is to:   1. Translate the coordinate system’s origin (0, 0) to where you want the upper left of the square to be. 2. Rotate the grid π/4 radians (45°) 3. Draw the square at the origin.   Grid translated, then rotated  And here is the code and its result, without the grid marks.  result of properly rotating square  void setup()  {  size(200, 200);  background(255);  smooth();  fill(192);  noStroke();  rect(40, 40, 40, 40);    pushMatrix();  // move the origin to the pivot point  translate(40, 40);    // then pivot the grid  rotate(radians(45));    // and draw the square at the origin  fill(0);  rect(0, 0, 40, 40);  popMatrix();  }  And here is a program that generates a wheel of colors by using rotation. The screenshot is reduced to save space.  multiply rotated rectangle in different colors  void setup() {  size(200, 200);  background(255);  smooth();  noStroke();  }  void draw(){  if (frameCount % 10 == 0) {  fill(frameCount \* 3 % 255, frameCount \* 5 % 255,  frameCount \* 7 % 255);  pushMatrix();  translate(100, 100);  rotate(radians(frameCount \* 2 % 360));  rect(0, 0, 80, 20);  popMatrix();  }  } Scaling The final coordinate system transformation is scaling, which changes the size of the grid. Take a look at this example, which draws a square, then scales the grid to twice its normal size, and draws it again.  gray square scaled up to double size  void setup()  {  size(200,200);  background(255);    stroke(128);  rect(20, 20, 40, 40);    stroke(0);  pushMatrix();  scale(2.0);  rect(20, 20, 40, 40);  popMatrix();  }  First, you can see that the square appears to have moved. It hasn’t, of course. Its upper left corner is still at (20, 20) on the scaled-up grid, but that point is now twice as far away from the origin as it was in the original coordinate system. You can also see that the lines are thicker. That’s no optical illusion—the lines really are twice as thick, because the coordinate system has been scaled to double its size.  **Programming Challenge:** Scale up the black square, but keep its upper left corner in the same place as the gray square. Hint: use translate() to move the origin, then use scale().  There is no law saying that you have to scale the *x* and *y* dimensions equally. Try using scale(3.0, 0.5) to make the *x* dimension three times its normal size and the *y* dimension only half its normal size. Order Matters When you do multiple transformations, the order makes a difference. A rotation followed by a translate followed by a scale will not give the same results as a translate followed by a rotate by a scale. Here is some sample code and the results.  result of different orders of rotate/translate/scale  void setup()  {  size(200, 200);  background(255);  smooth();  line(0, 0, 200, 0); // draw axes  line(0, 0, 0, 200);    pushMatrix();  fill(255, 0, 0); // red square  rotate(radians(30));  translate(70, 70);  scale(2.0);  rect(0, 0, 20, 20);  popMatrix();  pushMatrix();  fill(255); // white square  translate(70, 70);  rotate(radians(30));  scale(2.0);  rect(0, 0, 20, 20);  popMatrix();  } The Transformation Matrix Every time you do a rotation, translation, or scaling, the information required to do the transformation is accumulated into a table of numbers. This table, or *matrix* has only a few rows and columns, yet, through the miracle of mathematics, it contains all the information needed to do any series of transformations. And that’s why the pushMatrix() and popMatrix() have that word in their name. Push and Pop What, about the *push* and *pop* part of the names? These come from a computer concept known as a *stack*, which works like a spring-loaded tray dispenser in a cafeteria. When someone returns a tray to the stack, its weight pushes the platform down. When someone needs a tray, he takes it from the top of the stack, and the remaining trays pop up a little bit.  In a similar manner, pushMatrix() puts the current status of the coordinate system at the top of a memory area, and popMatrix() pulls that status back out. The preceding example used pushMatrix() and popMatrix() to make sure that the coordinate system was “clean” before each part of the drawing. In all of the other examples, the calls to those two functions weren’t really necessary, but it doesn’t hurt anything to save and restore the grid status.  Note: in Processing, the coordinate system is restored to its original state (origin at the upper left of the window, no rotation, and no scaling) every time that the draw() function is executed. Three-dimensional Transforms If you are working in three dimensions, you can call the translate() function with three arguments for the *x*, *y*, and *z* distances. Similarly, you can call scale() with three arguments that tell how much you want the grid scaled in each of those dimensions.  For rotation, call the rotateX(), rotateY(), or rotateZ() function to rotate around each of the axes. All three of these functions expect one argument: the number of radians to rotate. Case Study: An Arm-Waving Robot Let’s use these transformations to animate a blue robot waving its arms. Rather than try to write it all at once, we will do the work in stages. The first step is to draw the robot without any animation.  The robot is modeled on [this drawing](http://www.openclipart.org/detail/5457), although it will not look as charming. First, we draw the robot so that its left and top side touch the *x* and *y* axes. That will allow us to use translate() to easily place the robot anywhere we want or to make multiple copies of the robot, as we did in the example of the houses.  When we refer to left and right in this drawing, we mean your left and right (the left and right side of your monitor), not the robot’s left and right.  blue robot, arms at sides  void setup()  {  size(200, 200);  background(255);  smooth();  drawRobot();  }  void drawRobot()  {  noStroke();  fill(38, 38, 200);  rect(20, 0, 38, 30); // head  rect(14, 32, 50, 50); // body  rect(0, 32, 12, 37); // left arm  rect(66, 32, 12, 37); // right arm    rect(22, 84, 16, 50); // left leg  rect(40, 84, 16, 50); // right leg    fill(222, 222, 249);  ellipse(30, 12, 12, 12); // left eye  ellipse(47, 12, 12, 12); // right eye  }  robot with red dots at shoulder jointsThe next step is to identify the points where the arms pivot. That is shown in this drawing. The pivot points are (12, 32) and (66, 32). Note: the term “center of rotation” is a more formal term for the pivot point.  Now, separate the code for drawing the left and right arms, and move the center of rotation for each arm to the origin, because you always rotate around the (0, 0) point. To save space, we are not repeating the code for setup().  void drawRobot()  {  noStroke();  fill(38, 38, 200);  rect(20, 0, 38, 30); // head  rect(14, 32, 50, 50); // body  drawLeftArm();  drawRightArm();  rect(22, 84, 16, 50); // left leg  rect(40, 84, 16, 50); // right leg    fill(222, 222, 249);  ellipse(30, 12, 12, 12); // left eye  ellipse(47, 12, 12, 12); // right eye  }  void drawLeftArm()  {  pushMatrix();  translate(12, 32);  rect(-12, 0, 12, 37);  popMatrix();  }  void drawRightArm()  {  pushMatrix();  translate(66, 32);  rect(0, 0, 12, 37);  popMatrix();  }  Now test to see if the arms rotate properly. Rather than attempt a full animation, we will just rotate the left side arm 135 degrees and the right side arm -45 degrees as a test. Here is the code that needs to be added, and the result. The left side arm is cut off because of the window boundaries, but we’ll fix that in the final animation.  robot with arms at angle  void drawLeftArm()  {  pushMatrix();  translate(12, 32);  **rotate(radians(135));**  rect(-12, 0, 12, 37); // left arm  popMatrix();  }  void drawRightArm()  {  pushMatrix();  translate(66, 32);  **rotate(radians(-45));**  rect(0, 0, 12, 37); // right arm  popMatrix();  }  Now we complete the program by putting in the animation. The left arm has to rotate from 0° to 135° and back. Since the arm-waving is symmetric, the right-arm angle will always be the negative value of the left-arm angle. To make things simple, we will go in increments of 5 degrees.  int armAngle = 0;  int angleChange = 5;  final int ANGLE\_LIMIT = 135;  void setup()  {  size(200, 200);  smooth();  frameRate(30);  }  void draw()  {  background(255);  pushMatrix();  translate(50, 50); // place robot so arms are always on screen  drawRobot();  armAngle += angleChange;    // if the arm has moved past its limit,  // reverse direction and set within limits.  if (armAngle > ANGLE\_LIMIT || armAngle < 0)  {  angleChange = -angleChange;  armAngle += angleChange;  }  popMatrix();  }  void drawRobot()  {  noStroke();  fill(38, 38, 200);  rect(20, 0, 38, 30); // head  rect(14, 32, 50, 50); // body  drawLeftArm();  drawRightArm();  rect(22, 84, 16, 50); // left leg  rect(40, 84, 16, 50); // right leg    fill(222, 222, 249);  ellipse(30, 12, 12, 12); // left eye  ellipse(47, 12, 12, 12); // right eye  }  void drawLeftArm()  {  pushMatrix();  translate(12, 32);  rotate(radians(armAngle));  rect(-12, 0, 12, 37); // left arm  popMatrix();  }  void drawRightArm()  {  pushMatrix();  translate(66, 32);  rotate(radians(-armAngle));  rect(0, 0, 12, 37); // right arm  popMatrix();  } Case Study: Interactive Rotation Instead of having the arms move on their own, we will modify the program so that the arms follow the mouse while the mouse button is pressed. Instead of just writing the program at the keyboard, we first think about the problem and figure out what the program needs to do.  Since the two arms move independently of one another, we need to have one variable for each arm’s angle. It’s easy to figure out which arm to track. If the mouse is at the left side of the robot’s center, track the left arm; otherwise, track the right arm.  The remaining problem is to figure out the angle of rotation. Given the pivot point position and the mouse position, how do you determine the angle of a line connecting those two points? The answer comes from the atan2() function, which gives (in radians) the angle of a line from the origin to a given *y* and *x* coordinate. In constrast to most other functions, the *y* coordinate comes first. atan2() returns a value from -π to π radians, which is the equivalent of -180° to 180°.  But what about finding the angle of a line that doesn’t start from the origin, such as the line from (10, 37) to (48, 59)? No problem; it’s the same as the angle of a line from (0, 0) to (48-10, 59-37). In general, to find the angle of the line from (*x*0, *y*0) to (*x*1, *y*1), calculate  atan2(*y*1 - *y*0, *x*1 - *x*0)  Because this is a new concept, rather than integrate it into the robot program, you should write a simple test program to see that you understand how atan2() works. This program draws a rectangle whose center of rotation is its upper left corner at (100, 100) and tracks the mouse.  void setup()  {  size(200, 200);  }  void draw()  {  float angle = atan2(mouseY - 100, mouseX - 100);    background(255);  pushMatrix();  translate(100, 100);  rotate(angle);  rect(0, 0, 50, 10);  popMatrix();  }  That works great. What happens if we draw the rectangle so it is taller than it is wide? Change the preceding code to read rect(0, 0, 10, 50). How come it doesn’t seem to follow the mouse any more? The answer is that the rectangle really *is* still following the mouse, but it’s the short side of the rectangle that does the following. Our eyes are trained to want the long side to be tracked. Because the long side is at a 90 degree angle to the short side, you have to subtract 90° (or π/2 radians) to get the desired effect. Change the preceding code to read rotate(angle - HALF\_PI) and try it again. Since Processing deals almost exclusively in radians, the language has defined the constants PI (180°), HALF\_PI (90°), QUARTER\_PI (45°) and TWO\_PI (360°) for your convenience.  At this point, we can write the final version of the arm-tracking program. We start off with definitions of constants and variables. The number 39 in the definition of MIDPOINT\_X comes from the fact that the body of the robot starts at *x*-coordinate 14 and is 50 pixels wide, so 39 (14 + 25) is the horizontal midpoint of the robot’s body.  /\* Where upper left of robot appears on screen \*/  final int ROBOT\_X = 50;  final int ROBOT\_Y = 50;  /\* The robot's midpoint and arm pivot points \*/  final int MIDPOINT\_X = 39;  final int LEFT\_PIVOT\_X = 12;  final int RIGHT\_PIVOT\_X = 66;  final int PIVOT\_Y = 32;  float leftArmAngle = 0.0;  float rightArmAngle = 0.0;  void setup()  {  size(200, 200);  smooth();  frameRate(30);  }  The draw() function is next. It determines if the mouse is pressed and the angle between the mouse location and the pivot point, setting leftArmAngle and rightArmAngle accordingly.  void draw()  {  /\*  \* These variables are for mouseX and mouseY,  \* adjusted to be relative to the robot's coordinate system  \* instead of the window's coordinate system.  \*/  float mX;  float mY;    background(255);  pushMatrix();  translate(ROBOT\_X, ROBOT\_Y); // place robot so arms are always on screen  if (mousePressed)  {  mX = mouseX - ROBOT\_X;  mY = mouseY - ROBOT\_Y;  if (mX < MIDPOINT\_X) // left side of robot  {  leftArmAngle = atan2(mY - PIVOT\_Y, mX - LEFT\_PIVOT\_X)  - HALF\_PI;  }  else  {  rightArmAngle = atan2(mY - PIVOT\_Y, mX - RIGHT\_PIVOT\_X)  - HALF\_PI;  }  }  drawRobot();    popMatrix();  }  The drawRobot() function remains unchanged, but a minor change to drawLeftArm() and drawRightArm() is now necessary. Because leftArmAngle and rightArmAngle are now computed in radians, the functions don’t have to do any conversion. The changes to the two functions are in bold.  void drawLeftArm()  {  pushMatrix();  translate(12, 32);  **rotate(leftArmAngle);**  rect(-12, 0, 12, 37); // left arm  popMatrix();  }  void drawRightArm()  {  pushMatrix();  translate(66, 32);  **rotate(rightArmAngle);**  rect(0, 0, 12, 37); // right arm  popMatrix();  } |

# Object Oriented Programming

*This tutorial is for Processing version 1.1+. If you see any errors or have comments, please* [*let us know*](http://code.google.com/p/processing/issues/list)*. This tutorials is from the book,* [*Learning Processing*](http://www.processing.org/learning/books/#shiffman)*, by Daniel Shiffman, published by Morgan Kaufmann Publishers, Copyright 2008 Elsevier Inc. All rights reserved.*  
*Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*

## I'm Down with OOP

Before we begin examining the details of how object-oriented programming (OOP) works in Processing, let's embark on a short conceptual discussion of "objects" themselves. Imagine you were not programming in Processing, but were instead writing out a program for your day, a list of instructions, if you will. It might start out something like:

* Wake up.
* Drink coffee (or tea).
* Eat breakfast: cereal, blueberries, and soy milk.
* Ride the subway.

What is involved here? Specifically, what things are involved? First, although it may not be immediately apparent from how we wrote the above instructions, the main thing is you , a human being, a person. You exhibit certain properties. You look a certain way; perhaps you have brown hair, wear glasses, and appear slightly nerdy. You also have the ability to do stuff , such as wake up (presumably you can also sleep), eat, or ride the subway. An object is just like you, a thing that has properties and can do stuff.   
  
So how does this relate to programming? The properties of an object are variables; and the things an object can do are functions. Object-oriented programming is the marriage of all of the programming fundamentals: data and functionality.   
  
Let's map out the data and functions for a very simple human object:   
  
**Human data**

* Height.
* Weight.
* Gender.
* Eye color.
* Hair color.

**Human functions**

* Sleep.
* Wake up.
* Eat.
* Ride some form of transportation.

Now, before we get too much further, we need to embark on a brief metaphysical digression. The above structure is not a human being itself; it simply describes the idea, or the concept, behind a human being. It describes what it is to be human. To be human is to have height, hair, to sleep, to eat, and so on. Th is is a crucial distinction for programming objects. This human being template is known as a *class*. A *class* is different from an *object*. You are an object. I am an object. That guy on the subway is an object. Albert Einstein is an object. We are all people, real world *instances* of the idea of a human being.   
  
Think of a cookie cutter. A cookie cutter makes cookies, but it is not a cookie itself. The cookie cutter is the *class*, the cookies are the *objects*.

## Using an Object

Before we look at the actual writing of a class itself, let's briefly look at how using objects in our main program (i.e., **setup()** and **draw()**) makes the world a better place.   
  
Consider the pseudo-code for a simple sketch that moves a rectangle horizontally across the window (we'll think of this rectangle as a "car").   
  
**Data (Global Variables)**:

* Car color.
* Car x location.
* Car y location.
* Car x speed.

**Setup**:

* Initialize car color.
* Initialize car location to starting point.
* Initialize car speed.

**Draw**:

* Fill background.
* Display car at location with color.
* Increment car's location by speed.

To implement the above pseudo-code, we would define global variables at the top of the program, initialized them in setup(), and call functions to move and display the car in draw(). Something like:

color c = color(0);

float x = 0;

float y = 100;

float speed = 1;

void setup() {

size(200,200);

}

void draw() {

background(255);

move();

display();

}

void move() {

x = x + speed;

if (x > width) {

x = 0;

}

}

void display() {

fill(c);

rect(x,y,30,10);

}

Object-oriented programming allows us to take all of the variables and functions out of the main program and store them inside a car object. A car object will know about its data - *color*, *location*, *speed*. The object will also know about the *stuff it can do*, the methods (functions inside an object) - the car can *drive* and it can be *displayed*.   
  
Using object-oriented design, the pseudocode improves to look something like this:   
  
**Data (Global Variables)**:

* Car object.

**Setup**:

* Initialize car object.

**Draw**:

* Fill background.
* Display car object.
* Drive car object.

Notice we removed all of the global variables from the first example. Instead of having separate variables for car color, car location, and car speed, we now have only one variable, a Car variable! And instead of initializing those three variables, we initialize one thing, the Car object. Where did those variables go? They still exist, only now they live inside of the Car object (and will be defined in the Car class, which we will get to in a moment).   
  
Moving beyond pseudocode, the actual body of the sketch might look like:

Car myCar;

void setup() {

myCar = new Car();

}

void draw() {

background(255);

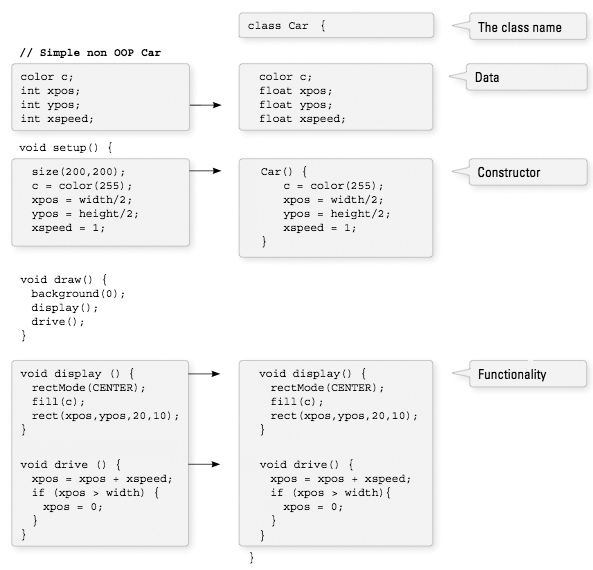
myCar.drive();

myCar.display();

}

We are going to get into the details regarding the above code in a moment, but before we do so, let's take a look at how the Car class itself is written.

## Writing the Cookie Cutter

The simple Car example above demonstrates how the use of object in Processing makes for clean, readable code. The hard work goes into writing the object template, that is the class itself. When you are first learning about object-oriented programming, it is often a useful exercise to take a program written without objects and, not changing the functionality at all, rewrite it using objects. We will do exactly this with the car example, recreating exactly the same look and behavior in an object-oriented manner.   
  
All classes must include four elements: name, data, constructor, and methods. (Technically, the only actual required element is the class name, but the point of doing object-oriented programming is to include all of these.)   
  
Here is how we can take the elements from a simple non-object-oriented sketch and place them into a Car class, from which we will then be able to make Car objects.   
  
  
  
**Class Name**: The name is specified by "class WhateverNameYouChoose". We then enclose all of the code for the class inside curly brackets after the name declaration. Class names are traditionally capitalized (to distinguish them from variable names, which traditionally are lowercase).   
  
**Data**: The data for a class is a collection of variables. These variables are often referred to as instance variables since each instance of an object contains this set of variables.   
  
**Constructor**: The constructor is a special function inside of a class that creates the instance of the object itself. It is where you give the instructions on how to set up the object. It is just like Processing's **setup()** function, only here it is used to create an individual object within the sketch, whenever a new object is created from this class. It always has the same name as the class and is called by invoking the new operator: "Car myCar = new Car();".   
  
**Functionality**: We can add functionality to our object by writing methods.   
  
Note that the code for a class exists as its own block and can be placed anywhere outside of **setup()** and **draw()**.

void setup() {

}

void draw() {

}

class Car {

}

## Using an Object: The Details

Earlier, we took a quick peek at how an object can greatly simplify the main parts of a Processing sketch (i.e. **setup()** and **draw()**).

**// Step 1. Declare an object.**

Car myCar;

void setup() {

**// Step 2. Initialize object.**

myCar = new Car();

}

void draw() {

background(255);

**// Step 3. Call methods on the object.**

myCar.drive();

myCar.display();

}

Let's look at the details behind the above three steps outlining how to use an object in your sketch.   
  
**Step 1. Declaring an object variable.**   
  
A variable is always declared by specifying a type and a name. With a primitive data type, such as an integer, it looks like this:

// Variable Declaration

int var; // type name

Primitive data types are singular pieces of information: an integer, a float, a character, etc. Declaring a variable that holds onto an object is quite similar. The diff erence is that here the type is the class name, something we will make up, in this case "Car." Objects, incidentally, are not primitives and are considered complex data types. (This is because they store multiple pieces of information: data and functionality. Primitives only store data.)   
  
**Step 2. Initializing an object.**   
  
In order to initialize a variable (i.e., give it a starting value), we use an assignment operation - variable equals something. With a primitive (such as integer), it looks like this:

// Variable Initialization

var = 10; // var equals 10

Initializing an object is a bit more complex. Instead of simply assigning it a value, like with an integer or floating point number, we have to construct the object. An object is made with the ***new*** operator.

// Object Initialization

myCar = new Car(); // The new operator is used to make a new object.

In the above example, "myCar" is the object variable name and "=" indicates we are setting it equal to something, that something being a new instance of a Car object. What we are really doing here is initializing a Car object. When you initialize a primitive variable, such as an integer, you just set it equal to a number. But an object may contain multiple pieces of data. Recalling the Car class, we see that this line of code calls the *constructor*, a special function named **Car()** that initializes all of the object's variables and makes sure the Car object is ready to go.   
  
One other thing; with the primitive integer "var," if you had forgotten to initialize it (set it equal to 10), Processing would have assigned it a default value, zero. An object (such as "myCar"), however, has no default value. If you forget to initialize an object, Processing will give it the value *null*. *null* means nothing. Not zero. Not negative one. Utter nothingness. Emptiness. If you encounter an error in the message window that says "NullPointerException" (and this is a pretty common error), that error is most likely caused by having forgotten to initialize an object.   
  
**Step 3. Using an object**   
  
Once we have successfully declared and initialized an object variable, we can use it. Using an object involves calling functions that are built into that object. A human object can eat, a car can drive, a dog can bark. Calling a function inside of an object is accomplished via dot syntax: variableName.objectFunction(Function Arguments);   
  
In the case of the car, none of the available functions has an argument so it looks like:

// Functions are called with the "dot syntax".

myCar.drive();

myCar.display();

## Constructor Arguments

In the above examples, the car object was initialized using the new operator followed by the constructor for the class.

Car myCar= new Car();

This was a useful simplification while we learned the basics of OOP. Nonetheless, there is a rather serious problem with the above code. What if we wanted to write a program with two car objects?

// Creating two car objects

Car myCar1 = new Car();

Car myCar2 = new Car();

This accomplishes our goal; the code will produce two car objects, one stored in the variable myCar1 and one in myCar2. However, if you study the Car class, you will notice that these two cars will be identical: each one will be colored white, start in the middle of the screen, and have a speed of 1. In English, the above reads:   
  
***Make a new car.***   
  
We want to instead say:   
  
***Make a new red car, at location (0,10) with a speed of 1.***   
  
So that we could also say:   
  
***Make a new blue car, at location (0,100) with a speed of 2.***   
  
We can do this by placing arguments inside of the constructor method.

Car myCar = new Car(color(255,0,0),0,100,2);

The constructor must be rewritten to incorporate these arguments:

Car(color tempC, float tempXpos, float tempYpos, float tempXspeed) {

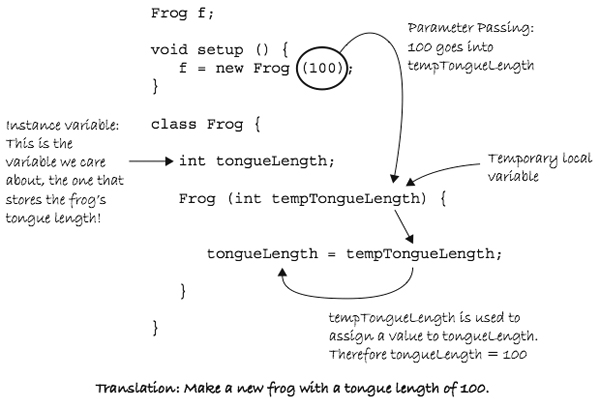
c = tempC;

xpos = tempXpos;

ypos = tempYpos;

xspeed = tempXspeed;

}

In my experience, the use of constructor arguments to initialize object variables can be somewhat bewildering. Please do not blame yourself. Th e code is strange-looking and can seem awfully redundant: "For every single variable I want argument to that constructor?"   
  
Nevertheless, this is quite an important skill to learn, and, ultimately, is one of the things that makes object-oriented programming powerful. But for now, it may feel painful. Let's looks at how parameter works in this context.   
  
  
  
Arguments are local variables used inside the body of a function that get filled with values when the function is called. In the examples, they have *one purpose only*, to initialize the variables inside of an object. These are the variables that count, the car's actual color, the car's actual *x* location, and so on. The constructor's arguments are just *temporary*, and exist solely to pass a value from where the object is made into the object itself.   
  
This allows us to make a variety of objects using the same constructor. You might also just write the word *temp* in your argument names to remind you of what is going on (c vs. tempC). You will also see programmers use an underscore (c vs. c\_) in many examples. You can name these whatever you want, of course. However, it is advisable to choose a name that makes sense to you, and also to stay consistent.   
  
We can now take a look at the same sketch with multiple object instances, each with unique properties.

[**// Example: Two Car objects**](http://www.learningprocessing.com/examples/chapter-8/example-8-2/)

Car myCar1;

Car myCar2; // Two objects!

void setup() {

size(200,200);

// Parameters go inside the parentheses when the object is constructed.

myCar1 = new Car(color(255,0,0),0,100,2);

myCar2 = new Car(color(0,0,255),0,10,1);

}

void draw() {

background(255);

myCar1.drive();

myCar1.display();

myCar2.drive();

myCar2.display();

}

// Even though there are multiple objects, we still only need one class.

// No matter how many cookies we make, only one cookie cutter is needed.

class Car {

color c;

float xpos;

float ypos;

float xspeed;

// The Constructor is defined with arguments.

Car(color tempC, float tempXpos, float tempYpos, float tempXspeed) {

c = tempC;

xpos = tempXpos;

ypos = tempYpos;

xspeed = tempXspeed;

}

void display() {

stroke(0);

fill(c);

rectMode(CENTER);

rect(xpos,ypos,20,10);

}

void drive() {

xpos = xpos + xspeed;

if (xpos > width) {

xpos = 0;

}

}

}

## Objects are data types too!

Assuming this is your first experience with object-oriented programming, it's important to take it easy. The examples here just one class and make, at most, two or three objects from that class. Nevertheless, there are no actual limitations. A Processing sketch can include as many classes as you feel like writing.   
  
If you were programming the Space Invaders game, for example, you might create a *Spaceship* class, an *Enemy* class, and a *Bullet* class, using an object for each entity in your game.   
  
In addition, although not primitive, classes are data types just like integers and floats. And since classes are made up of data, an object can therefore contain other objects! For example, let's assume you had just finished programming a *Fork* and *Spoon* class. Moving on to a *PlaceSetting* class, you would likely include variables for both a *Fork* object and a *Spoon* object inside that class itself. Th is is perfectly reasonable and quite common in object-oriented programming.

class PlaceSetting {

Fork fork;

Spoon spoon;

PlaceSetting() {

fork = new Fork();

spoon = new Spoon();

}

}

Objects, just like any data type, can also be passed in as arguments to a function. In the Space Invaders game example, if the spaceship shoots the bullet at the enemy, we would probably want to write a function inside the Enemy class to determine if the Enemy had been hit by the bullet.

void hit(Bullet b) {

// Code to determine if

// the bullet struck the enemy

}

When a primitive value (integer, float, etc.) is passed in a function, a copy is made. With objects, this is not the case, and the result is a bit more intuitive. If changes are made to an object after it is passed into a function, those changes will affect that object used anywhere else throughout the sketch. This is known as *pass by reference* since instead of a copy, a reference to the actual object itself is passed into the function.

# Upcoming changes in Processing 2.0 (revision 0198+)

As of June 18, 2011, we're making major changes in advance of Processing 2.0, which we would like to release in late summer or early fall. **As a result of these changes, using current builds straight from SVN is going to be messy!** If you want something more normal, check out a specific tagged release, i.e. processing-1.5.1, instead of using the trunk.

## Versions

We will be doing a series of "alpha" releases as we prepare for 2.0. Alpha means unstable and that function names and APIs will continue to change (mostly in PShape, XML, and other new bits like JSON and Table). It might be a bit like driving a sports car but with the hood removed and one of the tires might occasionally blow out. We recommend using these releases if you're not going to whine about quirks and having to update code along the way (we just gave you a goddamned sports car, for chrissake).

Sorry, where was I? Right, beta releases. After the alpha releases will be "beta", which means the APIs will stop changing, but the bugs might still be around. You'll have a proper set of tires and a hood, but you might still need a coat of paint and a radio. With any luck, the software will be better than my car analogies.

## The Big Stuff

* **P2D and P3D** have been removed. Using P2D will simply use the default (JAVA2D) renderer, and P3D will try to use OpenGL. We made this change to simplify things for us, and focus our efforts on better feature parity across other implementations. This fits better with the situation on Android and with JavaScript, for instance. This will cause some sketches to actually run slower, but the bottom line is that we simply don't have anyone to help maintain all of this extra code. We hope to sort out the performance problems over time—if you see something weird, please report a bug.
* **OpenGL 2** – a new version of the OpenGL library has been implemented, and the old one has been removed. The new library is based on Andres Colubri's Android work (and his experiences developing the GLGraphics library). All the great things from Android have now been back-ported to the desktop version of Processing, so we have a super fast OpenGL library. At the moment, the features that will provide the fastest performance gains (using PShape and VBOs) are not yet complete, however the overall performance is still significantly better.
* **Modes** – if you've used Processing 1.5, you'll know about the built-in Android mode, but if not, Processing now supports multiple languages and platforms. At the right-hand side of the editor window is a drop-down menu that allows you to choose between "Standard", "Android", and "JavaScript" mode. Those are the current modes that are being included, though we may add/remove modes as we head to 2.0 (a Jython mode is lurking about, for instance. Mmmm! Tempting.) Like Tools and Libraries, it will be possible for other parties to write their own modes that work inside the PDE.
* **JavaScript** – the JavaScript mode (see above) allows you to write a sketch and quickly run it in a browser using [Processing.js](http://processingjs.org). The mode was developed by long-time Processing contributor Florian Jenett, and continues to evolve. We highly recommend using JavaScript for running Processing work in web browsers.
* **Video** – we've removed the QuickTime for Java video library and are using a modified version of Andres Colubri's GSVideo library instead. On Linux, you'll need to install gstreamer to use the new library. On Windows and Mac OS X, you should not need to install it, however we're working out a few kinks in the whole process.
* **XMLElement** is gone, but we've added the XML class instead. With the change, you can call loadXML("blah.xml") from inside PApplet to read XML data. The rest of the API is the same as it was for XMLElement, except that getXxxxAttribute() is now just getXxxx(), for instance getIntAttribute() is just getInt() (to be more like the rest of the Processing API). Also added XML.parse(String) which returns an XML object from a String of XML data.

## Added and Improved

* Added **rounded rectangles** via rect(x, y, w, h, radius) and rect(x, y, w, h, tl, tr, br, bl).
* If you try to run more than one copy of Processing at a time, it will steadfastly refuse (and simply quit again). This isn't quite complete yet. Instead of just quitting the application it needs to bring the existing application to the front, and/or open the document you were trying to open with a new instance of Processing. But this will be sorted before 2.0 is released.

## Change and Removed

* **screen.width** and **screen.height** have been replaced by screenWidth and screenHeight. The originals were only half-documented and didn't properly fit with the rest of the Processing syntax.
* **delay()** has been removed. Nobody understood what it did, and when they did, they didn't understand why it was there. Huge source of confusion, especially for beginning students.
* **textMode(SCREEN)** has been removed. SCREEN was a super fast/efficient way of rendering text with P2D and P3D, but since they're going bye-bye and it's actually slower in the remaining renderers, it's going away.

## What's Coming

* A better interface for **installing and updating libraries, tools, and modes**. Our Google Summer of Code project will be getting more refined over time.
* Cleaner **32 and 64-bit support**. The underlying code for 32 and 64-bit libraries is in there, but we need to do more to allow you to select which one is in use, how the Java VM handles it, and controlling what mode you're running in.

*Downloaded on December 30, 2011 from www.processing.org for use at the SMU Faculty Workshop: Computational Art and Creative Coding with Processing.*