PROCESSING

An Introduction to Programming

Jeffrey L. Nyhoff

Larry R. Nyhoff

CRC Press
Taylor & Francis Group

A CHAPMAN & HALL BOOK

Processing: An Introduction to Programming

Processing: An Introduction to Programming

by
Jeffrey L. Nyhoff
Larry R. Nyhoff

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-1-4822-5595-9 (Paperback)

Library of Congress Cataloging-in-Publication Data

Names: Nyhoff, Jeffrey, author. | Nyhoff, Larry R., author.
Title: Processing : an introduction to programming / Jeffrey L. Nyhoff, Larry R. Nyhoff.
Description: Boca Raton : CRC Press, 2017.
Identifiers: LCCN 2016041238 | ISBN 9781482255959 (pbk. : alk. paper)
Subjects: LCSH: Processing (Computer program language) | Computer programming--Study and teaching.
Classification: LCC QA76.73.P75 N94 2017 | DDC 005.1071--dc23
LC record available at https://lccn.loc.gov/2016041238

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To Sharlene, Rebecca, Megan, and SaraKate.

Contents

Foreword

In the summer of 2008, I was invited to present at an NSF-sponsored CPATH workshop at La Salle University in Philadelphia. The title of the workshop was *Revitalizing Computer Science Education through the Science of Digital Media*. My first book on Processing had been recently published, and my talk introduced the Processing language and environment. I also discussed my own journey from painter to coder. One workshop attendee seemed especially animated during my talk, and his enthusiasm led to a very lively group discussion. This was my first encounter with Jeff Nyhoff.

Jeff like me is a hybrid, with a background in the arts (theater) and computing. Clearly, what excited Jeff during my talk was discovering another kindred spirit; at the time there were far less of us in academia (*at least out in the open*). Jeff and I are both fascinated by the connections we see between coding and arts practice; this intersection is now commonly referred to as "creative coding." Rather than seeing two radically disparate disciplines—sometimes generalized as extreme opposite, bifurcated left and right brain activities—we see a beautiful integration. But we also understand most (and probably far saner) people won't come to this conclusion on their own. Thus, we both work to spread the creative coding gospel, which has led to our common mission, and the ultimate purpose of this wonderful new book: radically change how computing is taught.

I'm not sure if the workshop was Jeff's initial exposure to Processing, but he instantly recognized its elegance and ultimate usefulness, especially for introductory computing education—something that its originators, Ben Fry and Casey Reas, didn't really consider outside of the arts context. Jeff and I remained in email contact after the workshop, and a few years later, at the annual computer science educators' conference (SIGCSE), we met again in person. It was there, wandering around the SIGCSE book publishers' exhibit, that I met Larry Nyhoff, Jeff's father. Larry was warm, cordial, and humble, and at first I didn't realize Jeff and Larry's professional connection beyond father and son. Through Jeff, Larry knew of me and my work, and too seemed excited by the possibilities Processing offered for the classroom. I left that encounter clueless that I had been speaking to a true computing educator legend.

Learning more about Larry gave me much greater insight into how Jeff, the theater major, became a computer science professor. Larry taught computer science for over 40 years at Calvin College and is the author or co-author of over 30 books, spanning many facets of computer science, as well as related (and seemingly unrelated) disciplines. Larry's books teach numerous programming languages and the aggregate of his publishing efforts

literally represents the history of modern computing education. The Processing literature has gotten a lot more respectable thanks to Larry's participation.

Jeff and Larry are deeply committed educators who have been willing to bring a critical, albeit loving, eye to their own discipline, especially around issues of programming pedagogy. I remember receiving many impassioned emails from Jeff discussing his excitement about Processing and CS pedagogy, but also his frustration with the established CS community's initial hesitancy adopting Processing and ultimately embracing change. It was inspiring for me to find someone so committed to helping students, literally *all students*, learn to code. This is why I was especially excited to learn Jeff and Larry were writing their own book on Processing.

The Nyhoffs' book arrives as Processing enters a very stable and mature form. This was certainly not the case for me when I began my book, back in 2004 when Processing was still a messy work in progress. Processing 3 is now a very stable, professional-grade code library and programming environment that is perfectly suited for use within the CS classroom and beyond. Yet, in spite of Processing's success and growing mainstream awareness, there are still very few Processing books designed specifically for the CS classroom. Processing's legacy is the arts and design community, and most of the existing Processing literature supports this population. The book *Creative Coding and Generative Art 2*, of which I am a co-author, was designed for the CS classroom, with a concentration on visual and graphic examples. This approach works well for some populations, but not all. And this is where Jeff and Larry's book comes to the rescue.

The Nyhoffs' new book directly targets the CS classroom in a way that no other Processing book does. The other books, including my own, unabashedly depart from traditional text-based examples found in almost all other introductory programming texts. In our defense, initiating change sometimes requires a small revolution. However, Jeff and Larry present a much less reactionary approach, integrating many of the wonderful things about Processing with traditional approaches that have worked well in CS pedagogy. Not only is their approach sensible and efficient, it's also likely to offer greater comfort to existing CS instructors (who perhaps don't have degrees in theater or painting).

It is this effort of considerate integration—of the *old tried and true* and *new and improved*—that I believe has the greatest chance of tipping the balance for Processing's use in the computing classroom.

Ira Greenberg
Dallas, Texas

Preface: Why We Wrote This Book and For Whom It Is Written

This book is based on our belief that Processing is an excellent language for beginners to learn the fundamentals of computer programming.

What Is Processing?

Processing was originally developed in 2001 at the MIT Media Lab by two graduate students, Ben Fry and Casey Reas, who wanted to make it simpler to use computer programming to produce visual art. Fry and Reas have since been joined by a community of developers who have continued to update and improve Processing. As an open-source software project, Processing is free for anyone to download, and versions are available for Windows, MacOS, and Linux computers.

Because Processing is based on Java, Processing is quite powerful. For example, if you explore such websites as https://processing.org and https://www.openprocessing.org, you'll see a wide variety of works that have been created using Processing: drawings, interactive art, scientific simulations, computer games, music videos, data visualizations, and much more. Some of these creations are remarkably complex.

Because Processing is based on Java, it has many similarities to Java. However, Processing is much easier to learn than Java. In fact, we believe that Processing might well be the *best* language currently available for teaching the basics of computer programming to beginners.

Why Not Java?

One of the most important features of Processing is that it enables us to begin the process of learning to program by building our programs out of simple statements and functions. Unlike strictly object-oriented languages such as Java, Processing does not require us to start with the more complex concepts and structures of "objects." This makes for what we believe is a much gentler introduction to programming. At the same time, because Processing is based on Java, we have the full capability to move on to object-oriented

programming whenever we are ready to do so. In fact, this book includes a basic introduction to objects.*

At the same time, because Processing is based on Java, it is usually very similar (and often identical) to Java in terms of much of its syntax and many of its structures, at least in regard to the topics typically covered in an introductory programming course. Thus, we have found that students proceed very smoothly from programming in Processing in a first course to programming in Java in a second course. In general, Processing's syntax and structures carry over readily to the learning of such programming languages as Java, C++, C#, and JavaScript, should you wish to learn any of these programming languages after learning Processing.

What about Python?

Python is becoming increasingly popular as an introductory programming language. However, Python often uses syntax and structures that are somewhat untraditional in comparison with other programming languages that a student might also wish to learn.

We have also found that the dynamic typing of variables in languages like Python is actually easier for students to understand *after* they have learned the static typing of variables in a language like Processing.†

Why This Book?

There are a number of very good books about Processing that have already been written. However, many of them seem to us to be geared first and foremost toward the creation of visual art. Processing certainly provides powerful capabilities for doing this, yet the sophistication of such impressive digital art sometimes comes at the cost of increased complexity in the code, potentially confusing a beginning programmer.

In this book, we focus on using Processing as a language to teach the basics of programming to beginners who may have any of a variety of reasons for wanting to learn to program. Thus, we do not attempt to do a wide survey of Processing's vast capabilities for graphical and interactive works. Rather, we use its capabilities for graphics and interactivity in order to create examples that we hope you will find to be simple, illustrative, interesting, and perhaps even fun. If your goal in learning to program is to create digital art, we believe that the foundation this book provides is an excellent place for you to begin and will prepare you well for other Processing books that are more oriented toward digital art. However, if you are seeking to learn to program for reasons other than the creation of digital art, rest assured that this book has also been written with you in mind.

It might seem surprising to some that a book about such a graphics-oriented programming language as Processing includes many examples that use numbers or text as the primary output. However, these are used for a variety of reasons. First, numerical and textual examples are sometimes easier to understand than graphical ones. Second, most

* Chapter 10 introduces objects.

† Processing does have a Python mode, Processing.py, for those who would like to learn to program in Python through Processing. There is also P5.js, a JavaScript "interpretation" of Processing.

graphical programs in Processing have a numerical basis that is often easier to understand in the form of text and numerical output. Third, some readers are likely to be learning to program for the purpose of being able to process numerical information (e.g., if they are interested in math, science, or business) and/or to process data that is likely to include textual information. Fourth, we hope that a combination of simple graphical examples with simple examples that use numbers and/or text will appeal to a broader range of learners. In our own experience, it is sometimes a visual example that first connects with a student, but at other times, it is a numerical and/or textual example that a student finds to be more illustrative. In still other cases, it might be a combination of these types of examples that successfully connects with a student.

We have followed what might be said to be a fairly *traditional* sequence of topics, one that we have found to work well for introducing programming. Also, we hope that such a sequence might seem comfortably familiar to instructors and to students who have undertaken some computer programming before. We have tried to introduce the key computer science concepts associated with introductory programming without going into such detail that risks being overwhelming. Supplementary materials will be available to instructors looking to introduce even more computer science concepts associated with the topics (e.g., in a "CS1" course). Processing's capabilities for animation and interactivity are not explored extensively in this book and are not touched upon until Chapter 7. However, additional examples using animation and interactivity are available for instructors who wish to provide more coverage of these features and introduce them earlier. In addition, several online chapters are provided that introduce slightly more advanced topics in Processing, such as two-dimensional arrays, manipulation of strings, and file input and output (processingnyhoff.com). Additional exercises for most of the chapters are also available.

Throughout the writing of this text, one of our primary concerns has been the *pace* of the material. Some might find it to be slower in comparison with most programming texts. However, in our experience, it can be easy to forget just how gradual a process is required for most beginners to learn to program. Our pace in this book is based on our experiences in teaching programming to a wide variety of beginners in a classroom. No prior programming experience is expected. We hope that you will find the pace of this book to be a good match to the rate at which you learn as well. It is our hope that this text will be useful to students and instructors in a first programming class, but we have tried to write this book in such a way that it will also be useful to persons teaching themselves to program.

Acknowledgments

Many thanks to Randi Cohen at CRC Press/Taylor & Francis Group for her support of this book and her expert guidance throughout the writing process.

Thanks also to Todd Perry at CRC Press/Taylor & Francis Group and Michelle van Kampen and her colleagues at Deanta Publishing Services for their production work on the editing, layout, and typesetting of this text and for their advice during our proofreading.

Thanks to Ira Greenberg for his Foreword and for his encouraging influence. It was one of his demonstrations of Processing that first convinced us of the enormous instructional potential of this programming language.

Thanks to the reviewers, who provided such helpful feedback in response to drafts of this book.

Thanks to faculty colleagues at Trinity for their encouragement and to the many Trinity students who have served as test subjects for most of the material in this book.

Thanks to our families and friends for supporting us in this venture.

Thanks to God for all blessings, including the opportunity to write this book.

Introduction: Welcome to Computer Programming

Congratulations on your decision to try computer programming!

Why Learn to Program?

It may be that programming a computer is something that you aren't sure you are able to do or even want to be able to do. If so, then we hope that this book changes your mind by providing you with a friendly, straightforward, and solid introduction to computer programming.

Being a computer "user" once meant writing programs for the computer. However, the emergence of personal computers beginning in the late 1970s was accompanied by a growth in the availability of "application" software, such as word processors and spreadsheet programs. Such software is written for "end users," who do not need to be computer programmers. Thus, the concept of a computer user as a programmer gradually shifted to that of someone who uses *software written by others*.

However, there are still some very good reasons for learning about computer programming in the current era of computing.

- **Learning computer programming teaches you about what software really is.** Computer software programs may seem complicated and mysterious. However, as you gain experience with simpler examples of computer software, you will start to understand the fundamental nature of computer software. This is important knowledge, given that more and more areas of our lives are affected by computer software.
- **Learning computer programming gives insight into what a programmer does.** Someday, you may find yourself working on a team that includes a computer programmer. Knowing something about what is involved in programming is likely to help you communicate and collaborate better with programmers.
- **Instead of changing to fit the software, you can design how the software works.** When we use software that other people have written, we are often forced to adapt our ways of thinking and working to the way that the software has been designed. In contrast, when you write your own software, you get to decide what the software does and how it works.

- **You cannot always count on someone else to create your software.** Computer software is needed by so many people and in so many ways that there can never be enough people with programming abilities to develop all the software that users would like to have. Thus, knowing enough about computer programming to create or customize software may help you to provide software that you or others need and would not otherwise be available.
- **Even a little knowledge of computer programming is better than none at all.** Many people have no experience at all with programming. Thus, having even a little knowledge of computer programming puts you a step ahead in terms of understanding computing.
- **Computer programming might already be, or turn out to be, something that you need to learn.** Many areas of study and careers are enriched by a basic knowledge of computer programming, and a growing number are starting to require it. Traditionally, programming has paired well with math, science, and business. But in the present era, interests and skills in arts and media also are excellent pairings with computer programming. Today, computer programs are used to solve a very wide range of problems and serve a wide variety of purposes in nearly every area of human society.
- **Computer programming is a creative activity.** Computer programs are often used to solve problems, but computer programming is also a creative activity. Thus, you might find yourself writing a program as a means of expressing yourself or simply out of an urge to create something new. More and more artists and designers are learning to program, especially those who work in electronic media.
- **Computer programming might turn out to be something that you are good at and enjoy doing.** You will never know this without giving programming a try!

These are just a few of the many good reasons for learning about computer programming.

What Is Programming?

Consider a "program" that you might be given at a band, orchestra, or choir concert. This program simply lists the order in which the musical numbers are performed. This notion of putting things in a certain **sequence** is one of the fundamental concepts of computer programming.

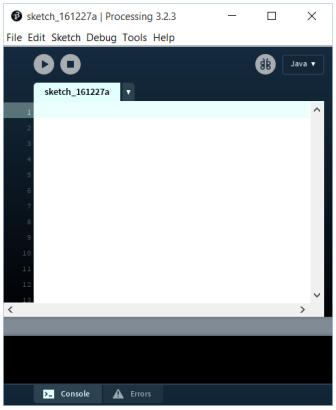
Another useful analogy is that of a recipe, where certain actions with various ingredients are performed in a certain order to achieve a certain result. A computer program is much like a recipe, involving the performance of a specific sequence of actions using certain "ingredients" (items of information) for the purpose of producing certain desired results. In computing, such a recipe is also known as an **algorithm**.

A Tour of Processing

We will give a more detailed introduction to Processing in the first chapter, but let's start here with a brief tour of Processing that will give you a little taste of what it's like and where we're headed in this book. *Do not worry* if you do not understand all the details that are presented in this tour; everything in this introduction will be explained more fully in the coming chapters.

Processing is an *open-source* project. It can be downloaded for free from **https://processing.org** and is available for Windows, Mac, and Linux computers.*

When you open Processing, what you see is Processing's integrated development environment or IDE:

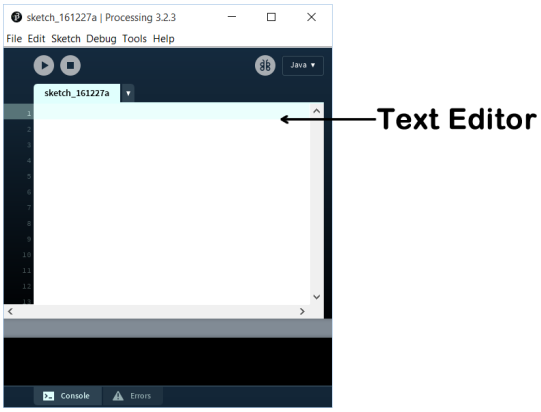


An IDE is simply a software environment in which you can develop and test programs. Compared with most IDEs, the Processing IDE is much simpler to use. Officially, the Processing IDE is known as the "Processing development environment" or PDE. Here in the PDE, you can enter programs and have Processing execute them.

"Hello, World!" Example

It is traditional in computer programming to begin with a program that simply displays the words "Hello, World!" Let's create such a program in Processing.

In the Text Editor section of the Processing window,

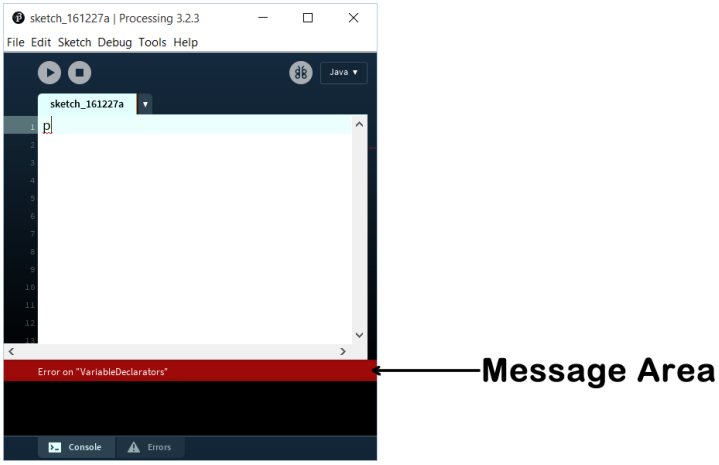


* As of this writing, the latest official release is version 3.3.

carefully enter the following:

```
print("Hello, World!");
```

For now, as you type, ignore any error messages that appear on a red bar in the Message Area of the Processing window:



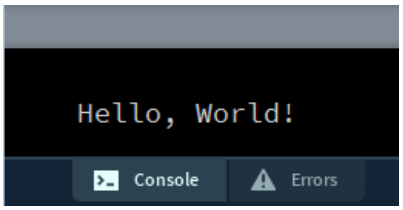
After you finish typing, these error messages should disappear. The top portion of the Text Editor area of the Processing window should now resemble the following:

```
1 print("Hello, World!");
2
```

You have now written a line of **code** in Processing. This particular line of code is also an example of what is called a **statement**, because it is a complete instruction to perform a certain action. When we run this program, Processing will carry out this instruction by performing the action it describes.

Next, click the **Run** button, which is found in the area of the Processing window known as the toolbar:

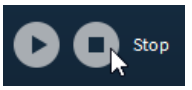
You should see "Hello, World!" displayed in the black section at the bottom of the Processing window. This section of the Processing window is known as the **console**.



If you don't see the above displayed in the console, then here are some things to check in your code:

- The word `print` must be written entirely in all lowercase letters.
- Make sure you have both a left and a right parenthesis.
- The message to be displayed, "Hello, World!" must be enclosed in a pair of double quotes.
- The statement you entered must end with a semicolon.

Once you see the "Hello, World!" greeting, you have successfully written and run your first program! You may now stop the program by pressing the **Stop** button:

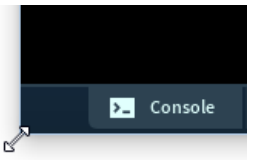


Notice the scroll bar on the right edge of the console:

This bar allows you to scroll through the lines of the console output.

There are occasions when it is helpful to enlarge the console area to see more of this program's output. To do this:

1) Click and drag on the edge of the Processing window to enlarge this window.



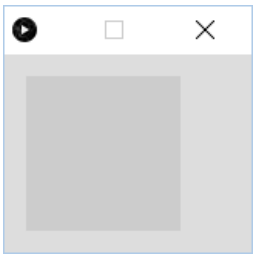
2) Click and drag upward on the Message Bar to enlarge the console.

You should now be able to view more lines of the console output without having to scroll.

A Processing "Sketch"

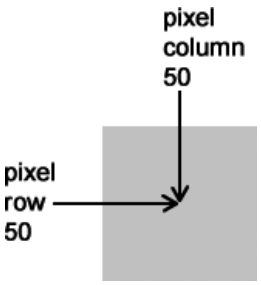
Displaying information in the console, such as our "Hello, World!" greeting, can be very useful when programming. However, there is a reason why programs in Processing are also called "sketches": programming in Processing can also involve *drawing*.

As you may have noticed, when you clicked on the Run button, Processing also opened what is called the **display window**. Within this window is a rectangular gray area that we will refer to as the **canvas** because it can be drawn on using Processing functions.



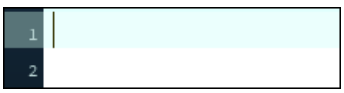
By default, a Processing sketch will render (generate) a canvas that consists of 100 columns of pixels and 100 rows of pixels. A **pixel** is a small square of a single color. In this case, all the pixels that make up the canvas are gray.

The columns of pixels are numbered from left to right, beginning with zero. Likewise, the rows of pixels are numbered from top to bottom, beginning with zero. Thus, each pixel has a unique location, described by the column and the row in which it is located. For example, the pixel in column 50 and row 50 is near the center of the current canvas.



Drawing an Ellipse

Let's delete our current statement that instructed Processing to display "Hello, World!" in the console. The Text Editor area of the Processing window should now be empty:



Now, enter the following into the Text Editor area of the Processing window:

```
ellipse(50, 50, 20, 10);
```

This is a statement instructing Processing to draw an ellipse. It includes four items of information that are needed to specify the ellipse that is to be drawn.

The first and second items of information specify where the *center* of the ellipse should be: the pixel column and the pixel row, respectively. In the current statement, we have chosen a pixel location of column 50 and row 50, the center of the canvas:

```
ellipse(50, 50, 20, 10);
```

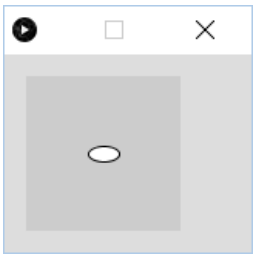
The third and fourth items specify the *size* of the ellipse to be drawn: the width of the ellipse in pixels and the height of the ellipse in pixels, respectively. In the current statement, we have chosen for the ellipse a width of 20 pixels and a height of 10 pixels:

```
ellipse(50, 50, 20, 10);
```

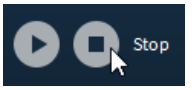
Notice that we have added a *space* after each comma. This spacing is not required, but this kind of "whitespace" can make code easier to read. In most cases, Processing is very forgiving of adding extra spaces and blank lines for this purpose.

Press the **Run** button.

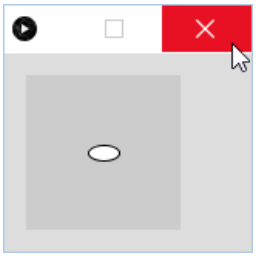
You should now see an ellipse with its center located at pixel column 50 and pixel row 50, with a width of 20 pixels and a height of 10 pixels:



You may now stop the program by pressing the **Stop** button:



Alternatively, you can click to close the display window:



Animation and Interactivity

The preceding ellipse-drawing program gave you a short demonstration of the kinds of programs we will ask Processing to execute. It is an example of a program written in what is called **static mode**. We will use static mode for much of this book because it allows us to write simpler programs.

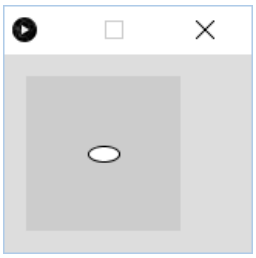
However, part of what makes Processing powerful and popular is its ability to create animated and interactive drawings. These more advanced capabilities won't be explored extensively in this book and are not introduced until Chapter 7 because they require us to do more advanced programming using Processing's **active mode** and `draw()` function. However, in the meantime, here is just a little taste of how these are possible in Processing. Again, *do not worry* if you do not understand everything that we are doing. For now, just type in the code and watch what happens.

Start by modifying your program so that it matches the following:

```
void draw()
{
 ellipse(50, 50, 20, 10);
}
```

What we have done here is to enclose our current program inside of what is called the `draw()` function. What is special about the `draw()` function is that it automatically "loops." In other words, Processing will automatically perform any statements that are contained within the `draw()` function over and over again when we run our program.

When we press the **Run** button, we see no change in our sketch:



This is because Processing is drawing an ellipse of the same size in the same canvas position, over and over again.

Press the **Stop** button.

Next, let's change the statement that we have placed within the `draw()` function to the following:

```
void draw()
{
 ellipse(mouseX, mouseY, 20, 10);
}
```

Here, we have replaced the specific numbers describing the pixel column and pixel row for the location of the center of the ellipse with `mouseX` and `mouseY`. (Notice that both

of these entries end in a capital letter: `mouseX` and `mouseY`.) These are two "variables" that are built into Processing and continuously keep track of the current pixel location of the mouse pointer when it is over the canvas. (We'll learn much more about variables in the chapters that follow.)

Now, press the **Run** button. If you move your mouse pointer over the canvas, you should see something like the following appear:

We see this result because, over and over again, Processing draws a 20-pixel by 10-pixel ellipse at the current location of the mouse pointer. You now have an *interactive* Processing sketch.

Hopefully, this brief tour gives you a taste of the nature of Processing. Again, this last animated and interactive sketch makes use of some advanced features that won't be covered until Chapter 7, so do not worry about not understanding all the details in this example. Rest assured, the features of Processing will be introduced to you thoroughly and gradually over the course of this book.

Exercises

1) Modify the "Hello, World!" example from this chapter so that when you run the program, your first name is displayed in the greeting instead.

   Example: `Hello, Rebecca!`

2) Modify the code from the "Hello, World!" example in this chapter to the following:

```
println("Hello, World");
println("Welcome to Processing!");
```

   Run the modified program to see what output is produced.

3) Modify the previous example so that it produces the following output:

```
Hello, World!
Welcome to Processing!
Let's get started.
```

4) Make the following change to the interactive sample program from this chapter:

```
void draw()
{
  ellipse(mouseX, mouseY, 10, 20);
}
```

Run this program and describe the change in the result.

CHAPTER 1

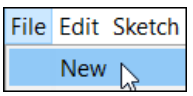
Basic Drawing in Processing

We'll now put aside the advanced capability associated with the looping `draw()` function and Processing's *active mode* that we explored at the end of the introduction. We will return instead to Processing's simpler *static mode* and work through some simple examples of basic drawing in Processing. These examples will also give us the opportunity to take a closer look at some of the fundamentals of computer programming in general.

This chapter is not an exhaustive introduction to Processing's extensive drawing capabilities. Rather, the goal is to give you enough of a foundation in drawing with Processing to understand the graphical examples in the chapters that follow.

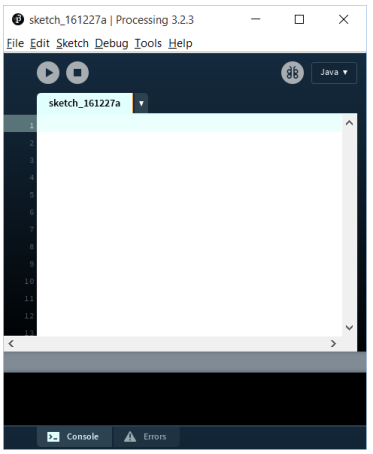
Starting a New Program

In the Processing window, pull down the **File** menu and select **New**.



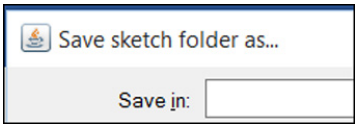
From now on, we will describe such a menu selection as **File** > **New**.

We now see a new Processing sketch window:



Saving a Program

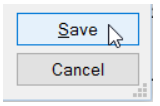
When starting a new program, it is a good idea to save it promptly and then to save it frequently after making changes to the program. "Save early and often!" is how this strategy is sometimes stated. This is good advice that can help to spare you from losing work that you have done. Saving a program in Processing is very much the same process as saving your work in other software that you use. Select **File** > **Save**. Since this is the first time you are saving this program, the **"Save sketch folder as…"** dialog box appears.



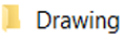
Using this dialog, start by choosing the file folder location where you would like to save the Processing program that you are currently writing. By default, Processing will save all your sketches (programs) in a file location that is known as your **Sketchbook**. This Processing program setting that specifies this default Sketchbook file location can be changed if you plan to save your Processing programs in a different location.* You can also just browse to the desired file folder location in the manner that you usually do when saving your work in other software programs that you use.

Next, in the **File name** textbox within the dialog box, enter a name for your program. It is customary to capitalize the first letter of a Processing program. Let's name this example **`Drawing`**.

Then, click on the **Save** button to finish saving this program.



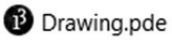
As a result, Processing creates a new **folder** in which your program is stored. This folder has the same name as that which you gave to your program. Thus, in this example, the new folder is named **`Drawing`**.



Processing calls this the **sketch folder**. If you examine the contents of a sketch folder, you will see a file that has the same name as the folder but also has the **`.pde`** file name extension. For example, in the current sketch folder, **`Drawing.pde`** is the name of the file containing our `Drawing` program.

---

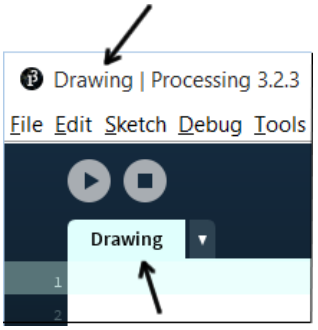
* You can change this default location of the Sketchbook by doing the following:
Select **File > Preferences**.
Click the **Browse** button next to the textbox labeled **Sketchbook Location**.
Browse to the folder where you would like your Processing programs saved.
Press the **Open** button.
Press the **OK** button. The folder you selected is now the default location where Processing will save and retrieve your programs.



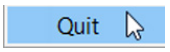
If you do *not* see the .pde portion of this file name, then you will need to turn on the viewing of file name extensions in your computer's operating system settings if you wish to see such file name extensions.*

Each .pde file created by Processing is simply a text file that stores all the lines of code that we write for a particular program.

Notice that once we save our program, the name we gave this program now appears both at the top of the Processing window and on a tab just above the Text Editor area:



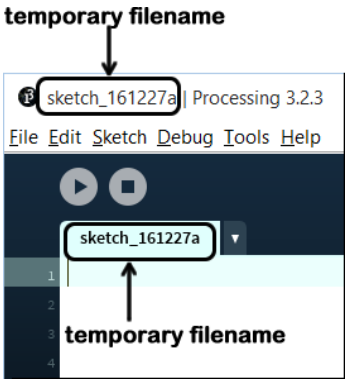
For now, close Processing by selecting **File > Quit**.



Retrieving a Program

Restart Processing.

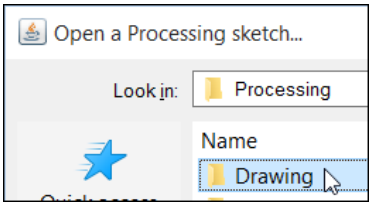
Notice that Processing always starts with a *new* program. Processing indicates that this new program has not yet been saved with a name by displaying a temporary name (based on the current date) in both the Processing window and the tab just above the editing area:



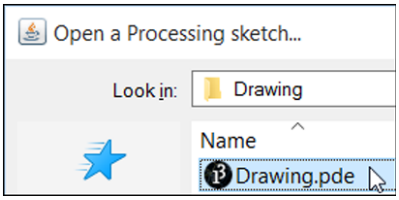
Here's a reliable way to retrieve a program that you have previously written and saved in Processing. Select **File > Open**. In the dialog that appears, locate the **`Drawing`** folder. (If

* For example, in Windows 10, simply open the File Explorer, click the View tab, and check the box labeled "File name extensions." For a Mac running OS X, open Finder > Preferences and then select "Show all filename extensions."

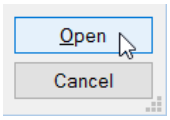
you saved your program in a location other than the default location, browse to that other location.)



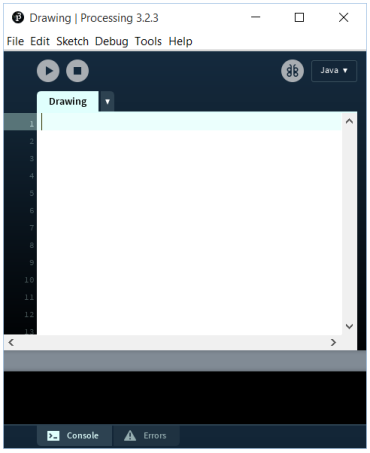
Double-click the `Drawing` folder to open it. You should now see the file named **`Drawing.pde`** that contains your program. (If you do not have file extensions turned on, then just `Drawing` will be shown as the name of this file.) Select this file.



Next, press the **Open** button.

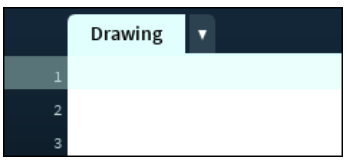


You should now see your `Drawing` program open once again in Processing:



Entering Code into the Text Editor

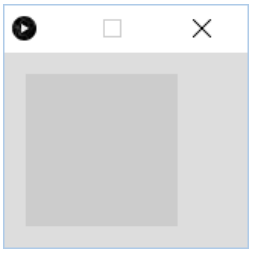
Our `Drawing` program does not yet contain any lines of code:



However, Processing still allows us to run this program. We can run a program by pressing the **Run** button.

Alternatively, we can select **File > Run**.*

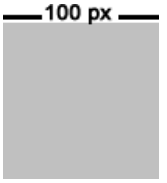
When we run our drawing program, we once again see the default canvas that Processing generates inside the display window:



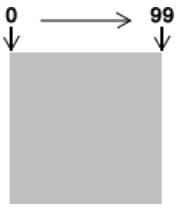
This canvas is a rectangular area that, by default, is comprised of columns and rows of gray pixels.

The word *pixel* is short for "picture element." This term reflects the fact that a pixel is simply a single-colored square that is but one piece of a larger image.†

As we learned previously, the **default width** of the Processing canvas is **100** columns of pixels:



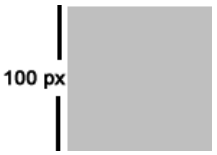
(*Pixels* is sometimes abbreviated *px*.) These pixel columns are numbered from left to right, starting at 0. Thus, for the default canvas, the pixel column numbers run from 0 to 99:



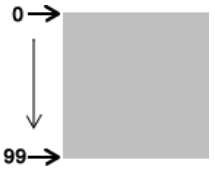
Likewise, the **default height** of the canvas is **100** rows of pixels.

* The file menu also shows a keyboard shortcut for running the program: Ctrl-R for Windows, Command-R (⌘-R) for Mac.

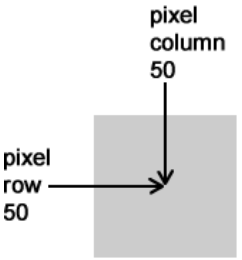
† This could be a piece of an image on the screen, a printed image, or an image in a file. The term *pixel* is used in all these contexts, which makes it a somewhat tricky term at times.



These rows of pixels are numbered from top to bottom, starting at 0. Thus, for the default canvas, the pixel row numbers also run from 0 to 99:



Accordingly, each pixel can be said to have a unique location, described by the column and row in which it is found. For example, the pixel near the center of the default canvas would be the pixel at column 50 and row 50:



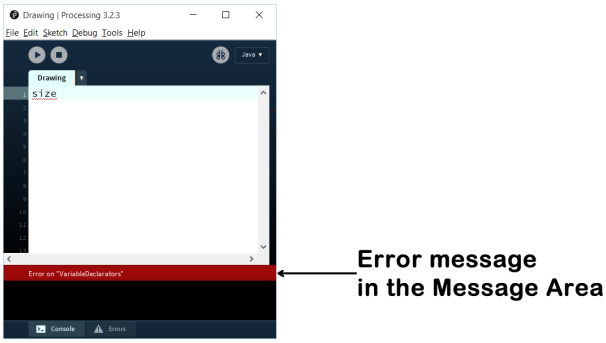
Processing has many built-in **functions** that are available for us to use. For example, during our "tour" of Processing, we used two such functions:

`print()` displays text in the console
`ellipse()` draws an ellipse on the canvas

In addition, the **`size()`** function lets us set the size of the canvas we are using. For example, let's enter the following statement into our program:

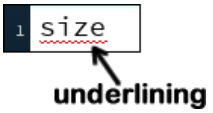
```
size(150, 200);
```

Notice that as you type, the area of the Processing window known as the Message Area display turns red and displays error messages:

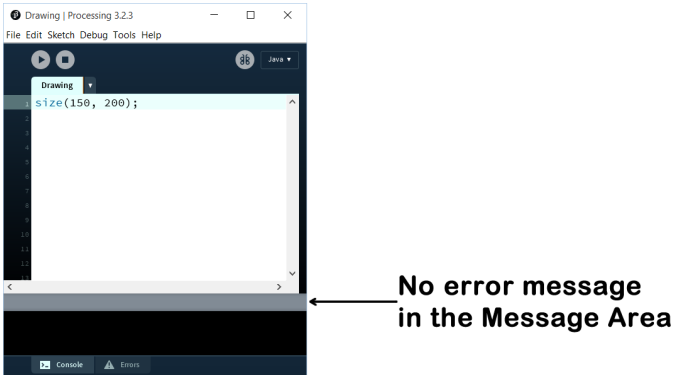


Do not be alarmed by this!

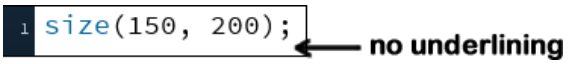
Also, as you type, certain portions of what you have entered may be underlined with a red squiggly line. Again, do not be alarmed by this.



After you have finished typing the line of code, the error messages should disappear from the Message Area:



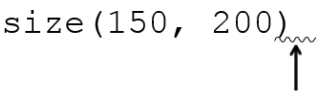
Also, the red squiggly line should disappear from beneath your line of code:



Such error messages and underlining by Processing might be a little confusing at first. However, while you are beginning to learn to program, it is a good habit simply to ignore these warnings from Processing until you have *completely entered* your line of code. If you have correctly entered your line of code, the warning messages and squiggly lines will disappear once you have finished entering it.

Once you become accustomed to these warnings, you will find that they actually can be quite useful. For example, try deleting the semicolon at the end of the line of code you just entered:

Notice that, now, a red squiggly line has appeared beneath the place in our line of code where we had previously entered a semicolon:



Notice also that an error message has once again appeared in the Message Area of the Processing window:

(Note: If you don't see this error message in the Message Area, then *click* on the squiggly red line with your mouse, and the error message will appear.) This is a useful error message. Most statements in Processing must end with a semicolon, and Processing is alerting us here that a semicolon is indeed required in this case. Processing is also warning us that if we try to run our program in its current state, we will receive an error. Most of Processing's error messages give us useful information like this.

As you program with Processing, you will occasionally encounter an error message that isn't entirely clear. However, one of the great improvements in recent versions of Processing is that the error messages have become much easier for beginning programmers to understand.

As soon as we correct our error by reinserting the semicolon

```
size(150, 200);
```

add the semicolon

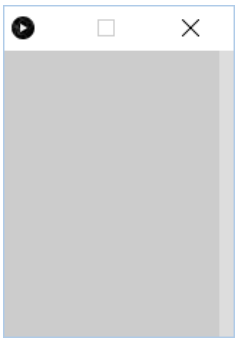
the squiggly line once again disappears, and the error message disappears from the Message Area.

Resave this `Drawing` program by selecting **File** > **Save**. We will continue working with this program in the next section.

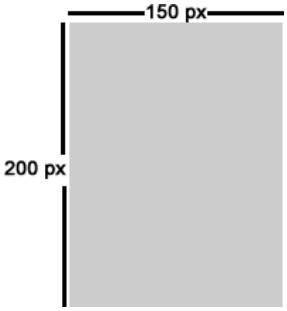
Basic Drawing with Graphical Elements

Setting the "Canvas" Size: A Closer Look at the `size()` Function

When we run our `Drawing` program again, we now see the following display window:



Within the display window is a canvas that is 150 pixel columns in width and 200 pixel rows in height.



Let's take a closer look at our use of the `size()` function. When we use a function in a statement like this, we say that we **call** this function. Like most functions, the `size()` function requires some information from us in order for it to be able to operate when we call it. We put such required information inside the parentheses that follow the name of the function. Such a required item of information is known as a **parameter**. The item supplied by us to fulfill this requirement is known as an **argument**. In other words, for each required *parameter*, we need to supply a corresponding *argument*.

The `size()` function has *two* required *parameters*, so we need to supply *two* corresponding *arguments*: the desired pixel *width* and the desired pixel *height*, listed in that order, separated by commas.

Thus, the basic form of a call to the `size()` function could be described in this way:

```
size(width, height);
```

The italicized words are the required parameters:

width	The desired pixel width of the canvas
height	The desired pixel height of the canvas

The word *syntax* is sometimes used in computer programming to describe the correct order and structure of the particular components that make up a larger element of the programming language. For example, the basic form of a call to the `size()` function is also known as that function's syntax.

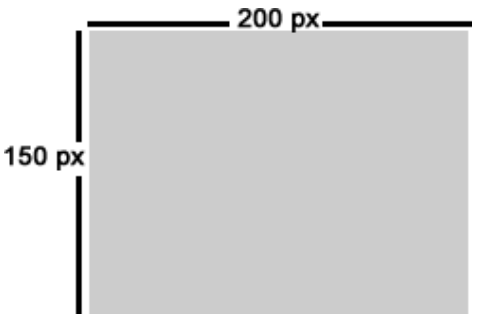
In our current program, we have supplied the following arguments in our call to the `size()` function, specifying the desired canvas *width* and canvas *height*, respectively:

```
size(150, 200);
```

It is important to notice that the *order* in which we supply these arguments is significant. For example, if we had reversed the numbers

```
size(200, 150);
```

then the specified canvas *width* would have been 200 columns of pixels, and the specified canvas *height* would have been 150 rows of pixels. These two arguments to the `size()` function would have specified a canvas with a different size:



Thus, make sure that the arguments in your call to the `size()` function appear in the correct order:

```
size(150, 200);
```

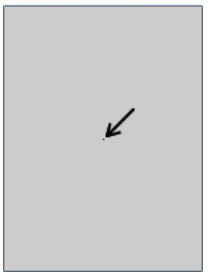
Select **File > Save** to resave the current program. We will continue working with this program in the next section.

Drawing Points: The `point()` Function

We can draw a "point" at a specific pixel location using the `point()` function. For example, let's add the following statement to our program:

```
size(150, 200);
point(75, 100);
```

When we run our program and look carefully, we see a single black pixel near the center of our canvas:



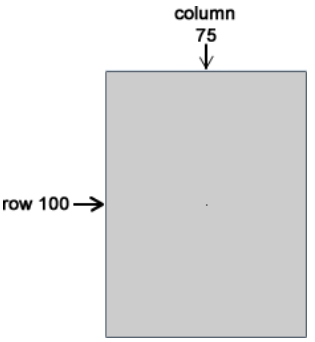
As we have seen, by default, Processing draws a point on the canvas as a single black pixel. Why was the point draw at this particular canvas location? Because the first argument we supply to the `point()` function

```
point(75, 100);
```

specifies the pixel *column* where the point is to be located. The second argument we supply to the `point()` function

```
point(75, 100);
```

specifies the pixel *row* where the point is to be located.



In summary, the basic form (syntax) of a statement making a call to the `point()` function is

```
point(column, row);
```

The parameters are

column	The pixel column where the point is to be drawn
row	The pixel row where the point is to be drawn

Our program is currently comprised of two statements:

```
size(150, 200);
point(75, 100);
```

It is very important to understand that when we run our program these statements are executed (performed) *by Processing in the order in which they are listed*:

1) `size(150, 200);` Generates a 150-pixel by 200-pixel blank canvas

2) `point(75, 100);` Draws a point at pixel column 75, pixel row 100

For example, if we changed the order of these statements to

```
point(75, 100);
size(150, 200);
```

then the order of execution would be

1) `point(75, 100);` Draws a point at pixel column 75, pixel row 100 on the *default* 100-pixel by 100-pixel canvas

2) `size(150, 200);` Generates a 150-pixel by 200-pixel blank canvas, *erasing* the point that was drawn

As a result, when we ran the program, we would see a blank canvas.

Thus, make sure your two statements in your program appear in the following order:

```
size(150, 200);
point(75, 100);
```

Select **File > Save** to resave this program. We will continue working with it in the next section.

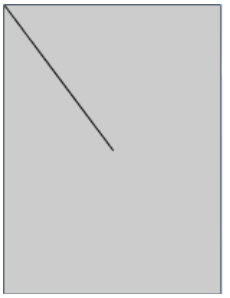
Drawing Line Segments: The `line()` Function

We can also draw line segments in Processing using the **`line()`** function. For example, let's change the second statement in our program to the following:

```
size(150, 200);
```

```
line(0, 0, 75, 100);
```

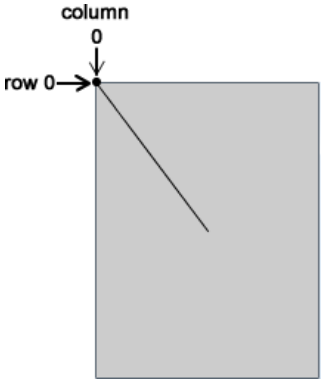
Now, when we run our program, we see the following:



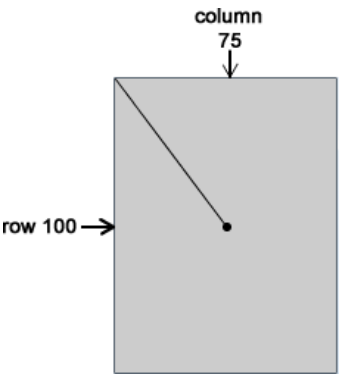
This particular line segment is drawn because the *first* and *second* arguments we supply to the `line()` function

```
line(0, 0, 75, 100);
```

specify the column and row, respectively, of the pixel location of one endpoint of the line segment:



Likewise, the *third* and *fourth* arguments we supply to the `line()` function specify the column and row, respectively, of the pixel location of the *other* endpoint of the line segment:



In summary, the basic form of a call to the `line()` function is

`line(column`$_1$`, row`$_1$`, column`$_2$`, row`$_2$`);`**

The parameters are

column$_1$	The pixel column location of the first endpoint
row$_1$	The pixel row location of the first endpoint
column$_2$	The pixel column location of the second endpoint
row$_2$	The pixel row location of the second endpoint

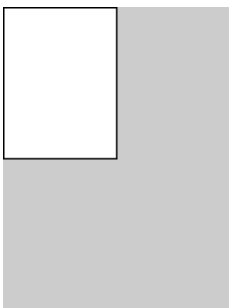
Thus, we simply supply the pixel locations of the endpoints of the line segment and Processing does the rest. It determines which pixels need to be recolored in order to generate the line segment we specify in the arguments of our call to the `line()` function. (By default, this is a line segment that is black and is one pixel in thickness.) This process by which Processing figures out the colors needed for each pixel of the canvas is sometimes known as **rendering**.

Drawing Rectangles: The `rect()` Function

Processing provides us with the ability to draw a rectangle on the canvas using the **`rect()`** function. For example, let's change the second statement in our program to the following:

```
size(150, 200);
rect(0, 0, 75, 100);
```

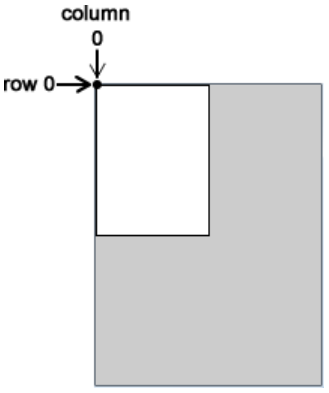
Now, when we run our program, we see the following:



Why does Processing render this particular rectangle? It's because the *first* and *second* arguments that we supply to the `rect()` function

```
rect(0, 0, 75, 100);
```

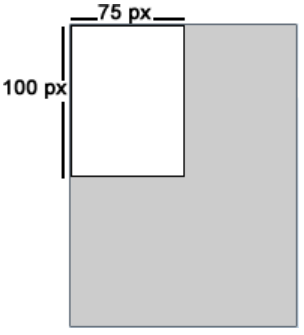
specify the *column* and *row*, respectively, of the pixel location of the *upper-left corner* of the rectangle:



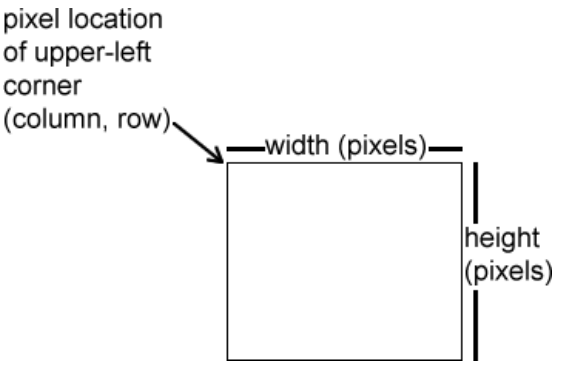
In addition, the *third* and *fourth* arguments that we supply to the `rect()` function

```
rect(0, 0, 75, 100);
```

specify the rectangle's pixel *width* and pixel *height*, respectively:



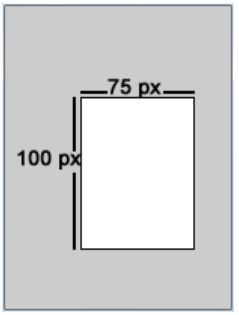
Thus, in general terms, we describe a rectangle with the `rect()` function by specifying the pixel location of its upper-left corner, its width, and its height:



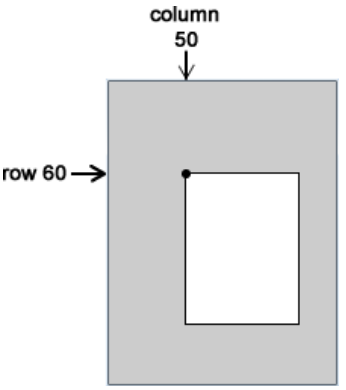
For example, if we change the *first* and *second* arguments in our call to the `rect()` function

```
size(150, 200);
rect(50, 60, 75, 100);
```

then, when we run our program, we see a rectangle that is the same size as before



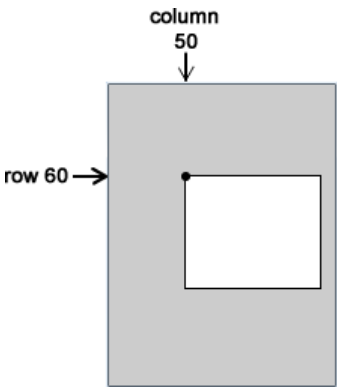
but its upper-left corner is now at pixel column 50 and pixel row 60:



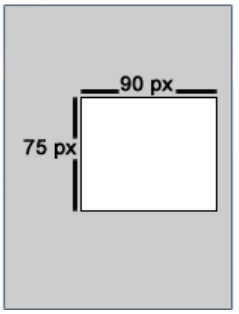
Likewise, if we change the *third* and *fourth* arguments in our call to the `rect()` function to the following,

```
size(150, 200);
rect(50, 60, 90, 75);
```

then, when we run our program, we still see a rectangle that has its upper-left corner at pixel column 50 and pixel row 60:



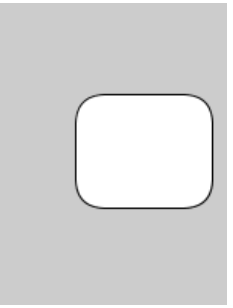
However, the rectangle's *width* is now 90 pixels, and its *height* is now 75 pixels:



Processing's `rect()` function also has an optional *fifth* parameter for drawing a *rounded* rectangle. For example, if we supply a fifth argument for an amount of rounding we would like to have applied to each corner of the rectangle,

```
size(150, 200);
rect(50, 60, 90, 75, 20);
```

then, when we run our program, we instead see a rounded rectangle:



In summary, the basic form of a call to the **`rect()`** function is

```
rect(column, row, width, height [, rounding] );
```

The parameters are

column	The pixel column location of the rectangle's upper-left corner
row	The pixel row location of the rectangle's upper-left corner
width	The width of the rectangle in pixels
height	The height of the rectangle in pixels
rounding	The amount of rounding of the corners (optional)

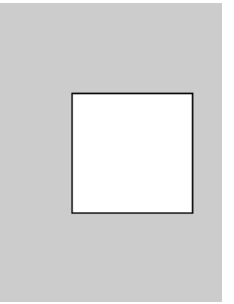
Thus, with these simple items of information from us, Processing is able to render a rectangle.

Notice that, by default, the edge of the rectangle is *black*. The color used for drawing points, lines, and the edges of shapes is known as the **stroke color**. Notice also that, by default, the rectangle is filled with *white* pixels. The color used to fill shapes is known as

the **fill color**. We will learn later in this chapter how to change both the stroke color and fill color.

There is not a function in Processing for drawing a *square*. However, we can easily draw a square using the `rect()` function if we simply specify the rectangle's width and height to be the same number of pixels:

```
size(150, 200);
rect(50, 60, 80, 80);
```



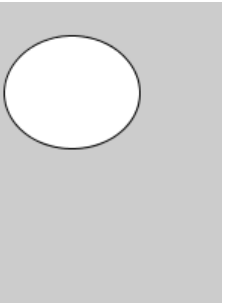
Select **File > Save** to resave this program. We will continue working with it in the next section.

Drawing Ellipses: The `ellipse()` Function

Processing also makes it easy for us to draw ellipses using the **`ellipse()`** function. For example, let's change the second statement in our program to the following:

```
size(150, 200);
ellipse(50, 60, 90, 75);
```

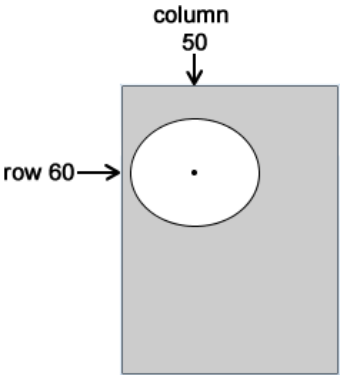
Now, when we run our program, Processing draws the following:



We see this particular ellipse because the *first* and *second* arguments we supply to the `ellipse()` function

```
ellipse(50, 60, 90, 75);
```

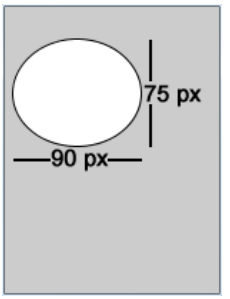
specify the *column* and *row*, respectively, of the pixel location of the *center* of the ellipse:



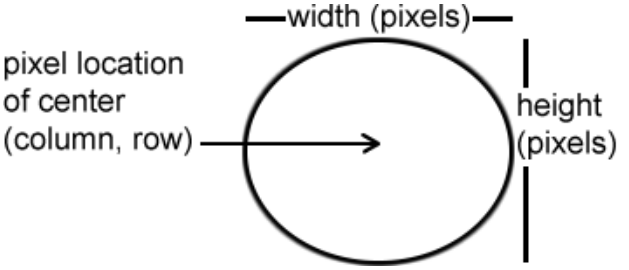
In addition, the *third* and *fourth* arguments that we supply to the `ellipse()` function

```
ellipse(50, 60, 90, 75);
```

specify the ellipse's pixel *width* and pixel *height*, respectively:



In summary, we describe an ellipse with the `ellipse()` function by specifying the pixel location of its center, its width, and its height:



Thus, the basic form of a call to the **`ellipse()`** function is

```
ellipse(column, row, width, height);
```

The parameters are

column	The pixel column location of the ellipse's center
row	The pixel row location of the ellipse's center

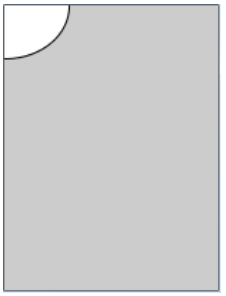
width	The width of the ellipse in pixels
height	The height of the ellipse in pixels

With these four simple items of information from us, Processing is able to render an ellipse.

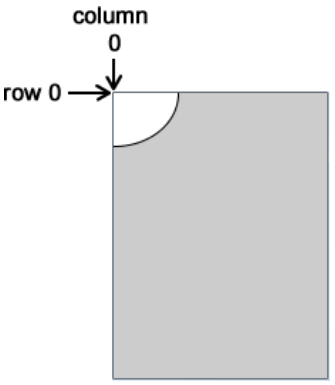
Now, let's try revising the second statement of our program so that we put the location of the center of the ellipse at pixel column 0 and pixel row 0:

```
size(150, 200);
ellipse(0, 0, 90, 75);
```

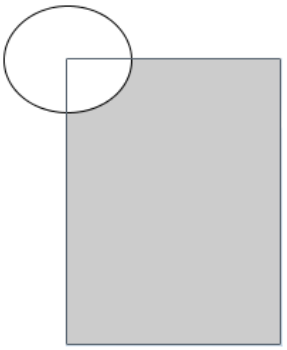
When we run our program, we now see the following:



We see this result because the center of our 90-pixel by 75-pixel ellipse is now located at pixel column 0 and pixel row 0, the pixel in the upper-left corner of the canvas:



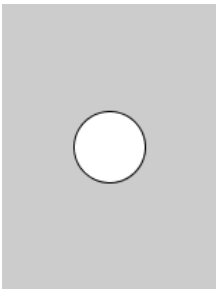
Thus, most of the ellipse we have specified essentially "spills" off the edge of the canvas.



Notice that Processing does not give us an error message when we instruct it to draw points, lines, or shapes that extend beyond the edges of the canvas. This provides a considerable degree of convenience and flexibility as we experiment with drawing. We are even permitted to use *negative* numbers when we wish to specify pixel columns and/or pixel rows that are outside the visible canvas.

Note that there is not a function in Processing for drawing a *circle*. However, we can easily draw a circle using the `ellipse()` function simply by specifying the ellipse's width and height to be the same number of pixels. For example, the following program draws a circle with a diameter of 50 pixels in the center of the canvas:

```
size(150, 200);
ellipse(75, 100, 50, 50);
```



Select **File > Save** to save this program. We will continue working with it in the next section.

Drawing Triangles: The `triangle()` Function

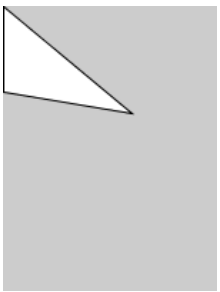
Let's learn about another basic shape function that Processing provides: the **`triangle()`** function.

The `triangle()` function allows us to draw a triangle simply by specifying *three* pixel locations on the canvas, each one corresponding to one of the three vertices (corner points) of that triangle.

For example, let's change the second line of our `Drawing` program to the following:

```
size(150, 200);
triangle(0, 0, 0, 60, 90, 75);
```

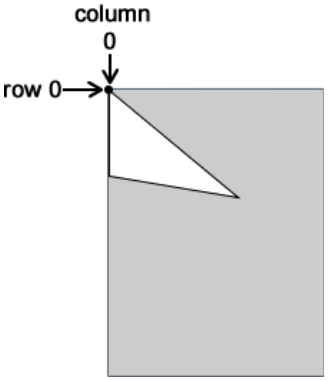
When we run our program, we see the following:



Processing draws this particular triangle because the *first* and *second* arguments we supply to the `triangle()` function

```
triangle(0, 0, 0, 60, 90, 75);
```

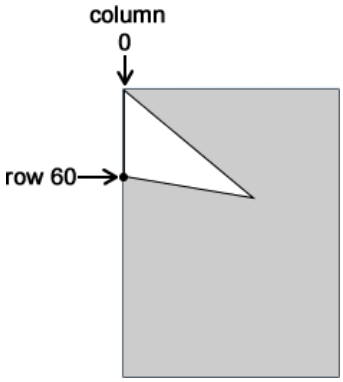
specify the *column* and *row*, respectively, of the *first* vertex (corner point) of the triangle:



Likewise, the *third* and *fourth* arguments we supply to the `triangle()` function

```
triangle(0, 0, 0, 60, 90, 75);
```

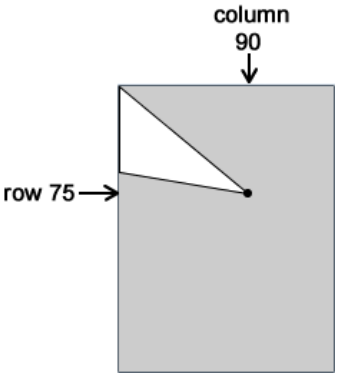
specify the *column* and *row*, respectively, of the *second* vertex of the triangle:



And, lastly, the *fifth* and *sixth* arguments we supply to the `triangle()` function

```
triangle(0, 0, 0, 60, 90, 75);
```

specify the *column* and *row*, respectively, of the *third* vertex of the triangle:



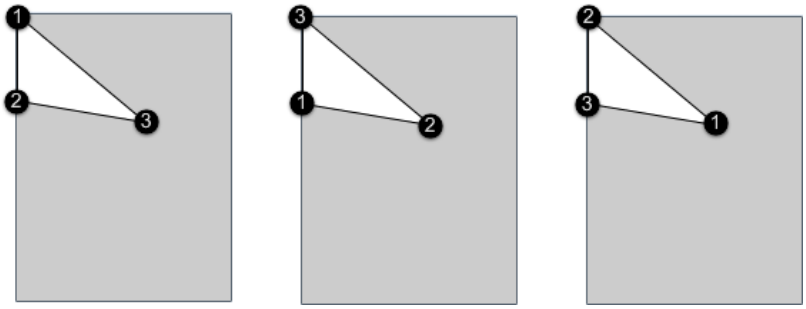
In summary, the basic form of the **`triangle()`** function is

```
triangle(column1, row1, column2, row2, column3, row3);
```

The parameters are

column$_1$	The pixel column location of the first vertex
row$_1$	The pixel row location of the first vertex
column$_2$	The pixel column location of the second vertex
row$_2$	The pixel row location of the second vertex
column$_3$	The pixel column location of the third vertex
row$_3$	The pixel row location of the third vertex

Note that the *same triangle* will be drawn regardless of the *order* in which the pixel locations of the three vertices are listed. In other words, it does not matter which of the three pixel locations is considered to be the first, second, or third vertex:



Select **File > Save** to save this program. We will continue working with it in the next section.

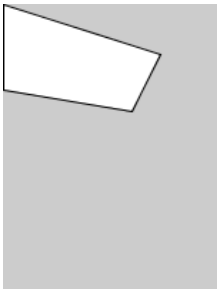
Drawing Quadrilaterals: The `quad()` Function

Processing's **`quad()`** function allows us to draw a quadrilateral simply by specifying *four* pixel locations on the canvas.

For example, if we modify the second statement of our current program to

```
size(150, 200);
quad(0, 0, 0, 60, 90, 75, 110, 35);
```

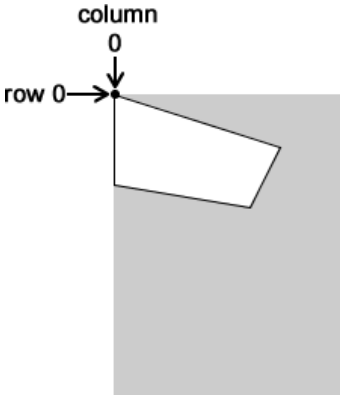
then, when we run our program, we see



Processing draws this particular quadrilateral because the *first* and *second* arguments we supply to the `quad()` function

```
quad(0, 0, 0, 60, 90, 75, 110, 35);
```

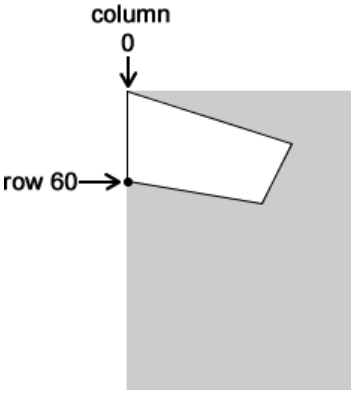
specify the *column* and *row*, respectively, of the pixel location for the *first* vertex of the quadrilateral:



The *third* and *fourth* arguments we supply to the `quad()` function

```
quad(0, 0, 0, 60, 90, 75, 110, 35);
```

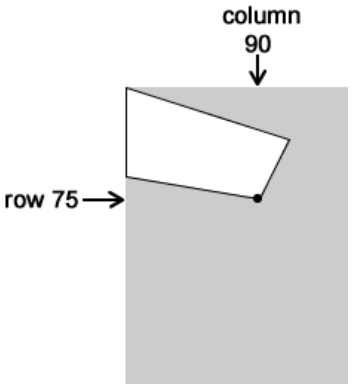
specify the *column* and *row*, respectively, of the pixel location for the *second* vertex of the quadrilateral:



The *fifth* and *sixth* arguments we supply to the `quad()` function

```
quad(0, 0, 0, 60, 90, 75, 110, 35);
```

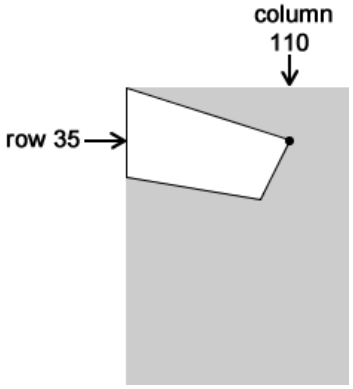
specify the *column* and *row*, respectively, of the pixel location for the *third* vertex of the quadrilateral:



The *seventh* and *eighth* arguments we supply to the `quad()` function

```
quad(0, 0, 0, 60, 90, 75, 110, 35);
```

specify the *column* and *row*, respectively, of the pixel location for the *fourth* vertex of the quadrilateral:



In summary, the basic form of the `quad()` function is

quad(*column*$_1$, *row*$_1$, *column*$_2$, *row*$_2$, *column*$_3$, *row*$_3$, *column*$_4$, *row*$_4$);

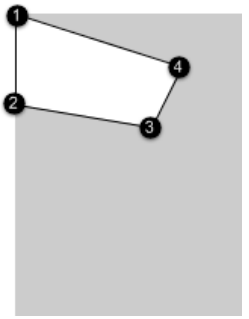
The parameters are

column$_1$	The pixel column location of the first vertex
row$_1$	The pixel row location of the first vertex
column$_2$	The pixel column location of the second vertex
row$_2$	The pixel row location of the second vertex
column$_3$	The pixel column location of the third vertex
row$_3$	The pixel row location of the third vertex
column$_4$	The pixel column location of the fourth vertex
row$_4$	The pixel row location of the fourth vertex

In the case of the `quad()` function, the *order* in which the vertices are listed *does* make a difference. For example, given the order in which the vertices are currently listed in the call to the `quad()` function in the second statement of our program,

```
size(150, 200);
quad(0, 0, 0, 60, 90, 75, 110, 35);
```

the order of our four vertices when the quadrilateral is drawn is considered to be



When the quadrilateral is drawn, the four line segments that are drawn connecting the four vertices are

From ❶ to ❷

From ❷ to ❸

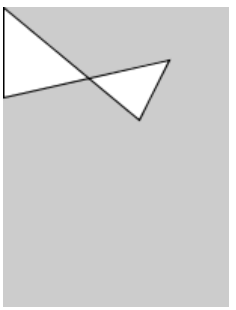
From ❸ to ❹

From ❹ to ❶

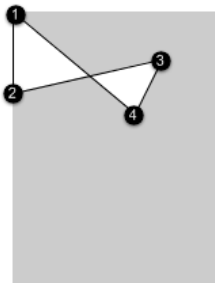
Accordingly, if we change the order of the vertices in our function call,

```
size(150, 200);
quad(0, 0, 0, 60, 110, 35, 90, 75);
```

then, when we run our revised program, we now see somewhat of a "bow tie" shape:



This is because the order of our vertices changed to



and the four line segments connecting the four vertices were drawn accordingly.

To avoid the bow tie effect, simply start the list of column and row arguments at a particular vertex and proceed in a clockwise (or counterclockwise) order until the column and row of all three of the other vertices have been listed.

Select **File > Save** to save this program. We will continue working with it in the next section.

Drawing Arcs: The `arc()` Function

Processing's **`arc()`** function allows us to draw *portions* of ellipses.

Recall that the basic form of the **`ellipse()`** function is

```
ellipse(column, row, width, height);
```

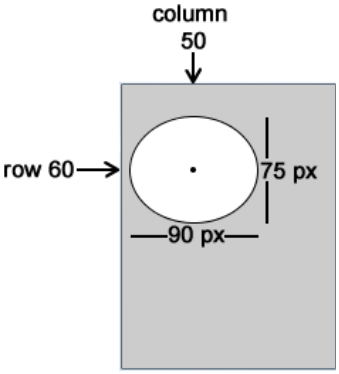
in which the four parameters are

`column`	The pixel column location of the ellipse's center
`row`	The pixel row location of the ellipse's center
`width`	The width of the ellipse in pixels
`height`	The height of the ellipse in pixels

Thus, the call to the **`ellipse()`** function in the following program,

```
size(150, 200);
ellipse(50, 60, 90, 75);
```

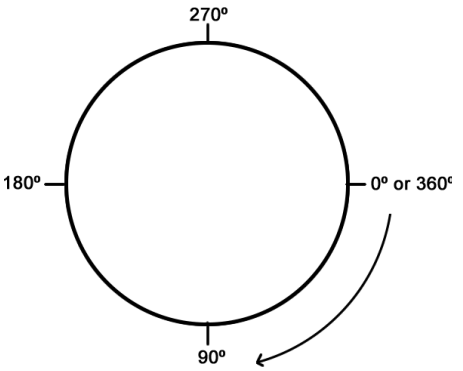
draws an ellipse with its center located at column 50 and row 60, a width of 90 pixels, and a height of 75 pixels:



The first four parameters of the **`arc()`** function are the same as those of the `ellipse()` function. However, there are two additional parameters, a **starting angle** and an **ending angle**, that specify the portion of the ellipse that is to be drawn.

The angle measures for the starting and ending angle must be given in *radians* rather than degrees. Fortunately, Processing provides a **`radians()`** function for converting degrees to radians.

It is also important to note that the angle measures proceed in a *clockwise* manner, the reverse of the counterclockwise direction you may have learned in math class.*

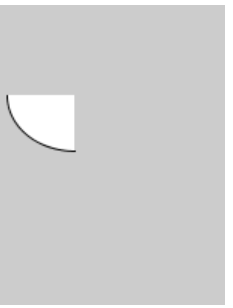


For example, let's change our call to the `ellipse()` function to the following call to the `arc()` function simply by changing the function name and by adding a starting angle and an ending angle, respectively:

```
size(150, 200);
arc(50, 60, 90, 75, radians(90), radians(180) );
```

Notice that we were able to specify our two angles using degrees, converting each angle measurement using the `radians()` function. As a result, there are now two calls to the `radians()` function *inside* our call to the `arc()` function. We are allowed to do this! Inserting a call to a function inside another call to a function like this is known as "nesting" function calls.

When we run our program, we now see the following result:

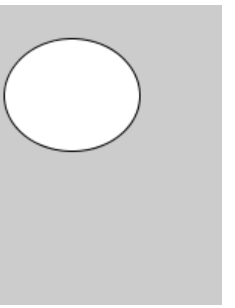


* This is because if we consider the columns and rows of pixels on the canvas to be like the *x* and *y* axes in geometry, then the *y* axis is "upside down" in comparison with geometry, because we move downward on the canvas rather than upward as the *y* values increase.

We see this result because the first four arguments,

```
arc(50, 60, 90, 75, radians(90), radians(180) );
```

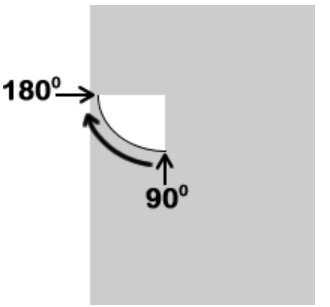
define an ellipse,



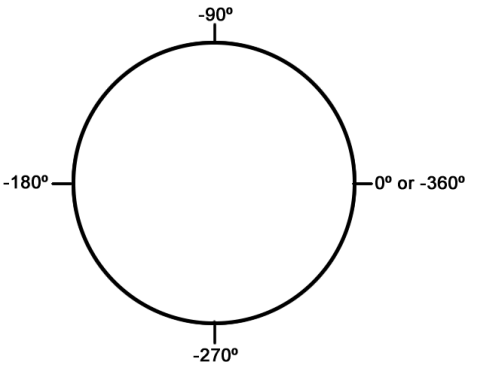
and the last two arguments,

```
arc(50, 60, 90, 75, radians(90), radians(180) );
```

specify that we want only the portion of this ellipse that is between the 90° position and the 180° position to be drawn:



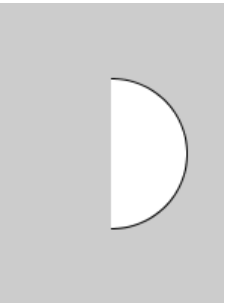
We can also describe the angles using *negative* degrees



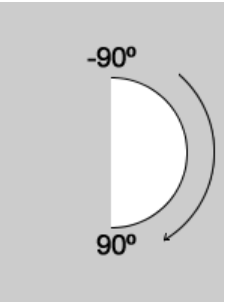
as long as the start angle and end angle describe a *clockwise* rotation. For example, the following call to the `arc()` function,

```
arc(50, 60, 90, 75, radians(-90), radians(90) );
```

will render the following arc:



We see this result because the starting angle was specified to be to be –90° and the ending angle was specified to be 90°. The use of a negative angle measure was allowed because the two angles still specified a clockwise rotation.



In summary, the basic form of the **`arc()`** function can be written as

```
arc(column, row, width, height, start, end);
```

in which the six parameters are

`column`	The pixel column location of the ellipse's center
`row`	The pixel row location of the ellipse's center
`width`	The width of the ellipse in pixels
`height`	The height of the ellipse in pixels
`start`	The angle at which the arc starts (given in radians)
`end`	The angle at which the arc ends (given in radians)

Select **File > Save** to resave the `Drawing` program.

Summary

In this section, we started by learning about a number of functions for drawing graphic elements in Processing.

size(*width*, *height*);

Parameters

width	The desired pixel width of the canvas
height	The desired pixel height of the canvas

point(*column*, *row*);

Parameters

column	The pixel column where the point is to be drawn
row	The pixel row where the point is to be drawn

line(*column*$_1$, *row*$_1$, *column*$_2$, *row*$_2$);

Parameters

column$_1$	The pixel column location of the first endpoint
row$_1$	The pixel row location of the first endpoint
column$_2$	The pixel column location of the second endpoint
row$_2$	The pixel row location of the second endpoint

rect(*column*, *row*, *width*, *height* [, *rounding*]);

Parameters

column	The pixel column location of the rectangle's upper-left corner
row	The pixel row location of the rectangle's upper-left corner
width	The width of the rectangle in pixels
height	The height of the rectangle in pixels
rounding	The amount of rounding of the corners (optional)

ellipse(*column*, *row*, *width*, *height*);

Parameters

column	The pixel column location of the ellipse's center
row	The pixel row location of the ellipse's center
width	The width of the ellipse in pixels
height	The height of the ellipse in pixels

triangle(*column*$_1$, *row*$_1$, *column*$_2$, *row*$_2$, *column*$_3$, *row*$_3$);

Parameters

column$_1$	The pixel column location of the first vertex
row$_1$	The pixel row location of the first vertex
column$_2$	The pixel column location of the second vertex
row$_2$	The pixel row location of the second vertex
column$_3$	The pixel column location of the third vertex
row$_3$	The pixel row location of the third vertex

quad(*column*$_1$, *row*$_1$, *column*$_2$, *row*$_2$, *column*$_3$, *row*$_3$, *column*$_4$, *row*$_4$);

Parameters

column$_1$	The pixel column location of the first vertex
row$_1$	The pixel row location of the first vertex

column$_2$	The pixel column location of the second vertex
row$_2$	The pixel row location of the second vertex
column$_3$	The pixel column location of the third vertex
row$_3$	The pixel row location of the third vertex
column$_4$	The pixel column location of the fourth vertex
row$_4$	The pixel row location of the fourth vertex

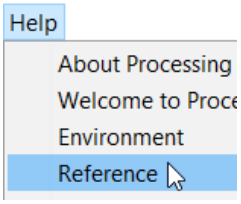
```
arc(column, row, width, height, start, end);
```

Parameters

column	The pixel column location of the ellipse's center
row	The pixel row location of the ellipse's center
width	The width of the ellipse in pixels
height	The height of the ellipse in pixels
start	The angle at which the arc begins (given in radians)
end	The angle at which the arc ends (given in radians)

The Processing Reference

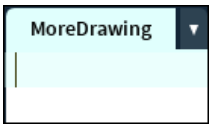
Additional information about all the above functions can be found in the **Reference** that is built into Processing and can be reached via the **Help** menu:



Some of the information is rather technical, so do not be worried if you don't understand all that is written in the Reference!

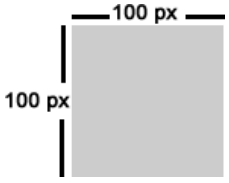
More about Graphical Elements

Let's learn more about working with graphical elements in Processing. Select **File > New** to start a new program. Select **File > Save** and save this new program as **`MoreDrawing`**.



(You may **close** the *other* Processing window containing the `Drawing` program.)

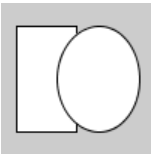
Our `MoreDrawing` program doesn't yet contain any lines of code. Thus, when we run it, we see only a display window containing a blank gray canvas of the default size, 100 pixels by 100 pixels:



In the Processing window containing the `MoreDrawing` program, enter the following two lines of code:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

When we run this program, we see that Processing simply draws a rectangle and an ellipse on a canvas of the default size (100 pixels by 100 pixels):



Select **File > Save** to resave this `MoreDrawing` program.

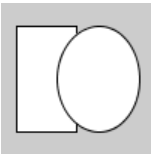
Throughout this section, we will repeatedly return to these two lines of code as our starting point.

Stacking Order

Notice that when we run our current program,

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

the ellipse is drawn *on top of* the rectangle.

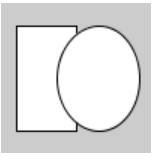


We see this result because of what is sometimes known as the **stacking order**. In Processing, when a graphical element is drawn *after* some other graphical element(s), it is drawn *on top of* those graphical elements that were drawn previously.

In our current program, the call to the `rect()` function is the *first* statement in the program, and the call to the `ellipse()` function is the *second* statement of the program:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

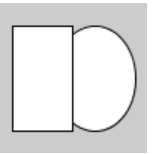
Therefore, when we run this program, the ellipse is drawn *after* the rectangle. Accordingly, the ellipse is drawn *on top of* the rectangle.



On the other hand, suppose we *reverse* the order of the statements in our program:

```
ellipse(65, 50, 55, 70);
rect(10, 15, 40, 70);
```

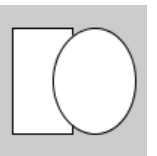
Now, when we run our program, the rectangle is drawn *after* the ellipse and, as a result, the rectangle is drawn *on top* of the ellipse:



Let's return our two statements to their original order:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

When we run the program, the ellipse is once again drawn on top of the rectangle:



In some graphic design software, it is possible to change the stacking order after the graphical elements are drawn. However, in Processing, the only way to change the stacking order of graphical elements is by changing the order in which they are drawn.

This example illustrates a very important principle in computer programming: the performing of actions in a specified **sequence**. As is the case in a cooking recipe, the order in which the individual actions that make up a computer program are performed is significant. Changing the order of the actions may produce a different result.

Changing Line Thickness: The `strokeWeight()` Function

By default, the points, the line segments, and the edges around the shapes we have drawn have all been one pixel in thickness. This attribute of these graphical elements is known in Processing as the **stroke weight**.

Processing provides the **`strokeWeight()`** function that enables us to change the stroke weight used when points, line segments, or shape edges are subsequently drawn.

The general form of this function is

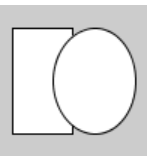
```
strokeWeight(pixels);
```

The only parameter of the `strokeWeight()` function is

`pixels` The desired stroke weight, specified in pixels

Consider once again our current program:

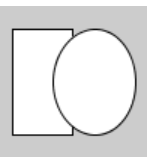
```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```



If we add the statement

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
strokeWeight(10);
```

then, when we run this program, we see the same result as before:



There is no change to the drawing that Processing produces because our call to the `strokeWeight()` currently take place *after* the calls to the `rect()` and `ellipse()` functions. In other words, we are changing the stroke weight too late, after the rectangle and ellipse have already been drawn. Only a point, line segment, or shape edge drawn *after* a call to the `strokeWeight()` function will have this new stroke weight.

Let's reposition the call to the `strokeWeight()` function so that it is *before* the call to the `ellipse()` function but *after* the call to the `rect()` function:

```
rect(10, 15, 40, 70);
strokeWeight(10);
ellipse(65, 50, 55, 70);
```

Now, when we run our program, only the ellipse is drawn with this new stroke weight:

On the other hand, if we position the call to the `strokeWeight()` function before *both* the call to the `rect()` function and the call to the `ellipse()` function,

```
strokeWeight(10);
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

then, when we run our program, *both* the rectangle and the ellipse are drawn with this new stroke weight:

There is an important general principle at work here: when we change a setting for the drawing of a graphical element, that change in the setting will *remain in effect* while we draw additional graphical elements. This is why *both* the rectangle and the ellipse were drawn with the new stroke weight: the current stroke weight remains in effect until it is reset using a new call to the `strokeWeight()` function.

If we would like the outline of the ellipse to have a different stroke weight, then we can simply insert a second call to the `strokeWeight()` function that comes *after* the call to the `rect()` function but *before* the call to the `ellipse()` function:

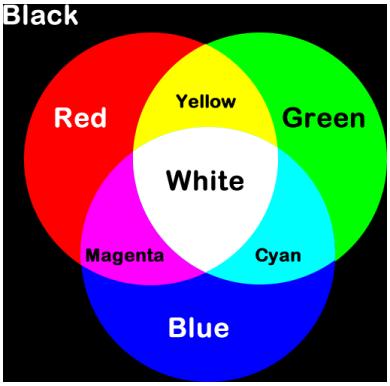
```
strokeWeight(10);
rect(10, 15, 40, 70);

strokeWeight(5);
ellipse(65, 50, 55, 70);
```

Now, when we run our program, the rectangle and the ellipse have different stroke weights:

Working with Color: RGB

So far, we have used only three pixel colors when drawing in Processing: black, white, and a shade of gray. However, Processing allows us to draw with colors as well.

Colors must be described as *numbers* in order for computers to be able to work with them. One of the most common ways that colors are numbered is according to the **RGB** color model, where all colors are described as mixed amounts red (R), green (G), and blue (B) light.



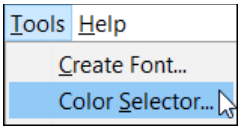
The three primary colors of light are red, green, and blue. When red light is mixed with green light, yellow light is produced. When red light is mixed with blue light, magenta light is produced. When green light is mixed with blue light, cyan light is produced. When red, green, and blue light are mixed in maximum amounts, white light is produced. When there is no red, no green, and no blue light at all, black is the result.

When the RGB model is used in computer technologies, the amounts of red, green, and blue light are typically described on a scale from 0 to 255. Thus, any color is described in RGB as a set of three numbers, each of which is from 0 to 255. For example, the RGB numbers for the basic colors in the above diagram are the following:

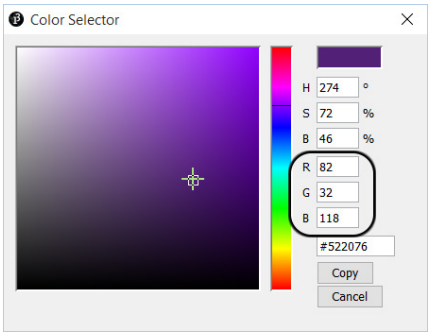
	Red	Green	Blue
Red	255	0	0
Green	0	255	0
Blue	0	0	255
Yellow	255	255	0
Magenta	255	0	255
Cyan	0	255	255
White	255	255	255
Black	0	0	0

Why 255? The amount of each of the red, green, and blue components is actually an eight-digit binary number, which is also known as 1 "byte." The largest binary number that can be stored in a byte is 11111111, which is the number we write in decimal as 255.

Processing also provides a built-in tool for determining the RGB numbers of a particular color. From the **Tools** menu, select **Color Selector**:



In the window that appears, we can click to select a certain color, and its RGB values will be displayed in the corresponding textboxes:



Resetting the Canvas: The `background()` Function

As we have seen, the default color of the canvas is a particular shade of gray. However, Processing provides the **`background()`** function, which resets all the pixels of the canvas to whatever RGB color we specify.

The basic form of a call to the `background()` function is

```
background(red, green, blue);
```

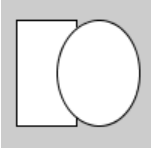
The parameters are

red	The amount of red (0 to 255)
green	The amount of green (0 to 255)
blue	The amount of blue (0 to 255)

Let's return to the starting version of our `MoreDrawing` program by removing the statements other than the following:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

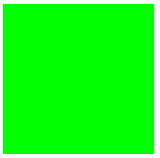
When we run this program, we once again see the following:



To change the background to green pixels, we might try the following:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
background(0, 255, 0);
```

However, when we run this program, we see a canvas that consists entirely of green pixels:



Why is this the result? As it turns out, the `background()` function is somewhat misnamed. Instead of changing only the background pixels, the `background()` function actually resets *all* the pixels on the canvas to the specified color. For this reason, we need to set the background color of the canvas *before* we draw on the canvas:

```
background(0, 255, 0);
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

When we run this program, we now see the rectangle and ellipse drawn on top of a green canvas:

Changing the Fill Color: The `fill()` and `noFill()` Functions

By default, Processing fills the shapes that we draw—ellipse, rectangles, triangles, and quadrilaterals—with white pixels. This is known as the **fill color**.

Processing provides the **`fill()`** function, which allows us to change the fill color of any shape that we draw *after* we call this function.

The basic form of a call to the **`fill()`** function is

```
fill(red, green, blue);
```

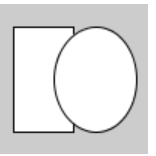
The parameters are

red	The amount of red (0 to 255)
green	The amount of green (0 to 255)
blue	The amount of blue (0 to 255)

Let's once again return to the starting version of our `MoreDrawing` program by deleting all of the lines of code other than the following:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

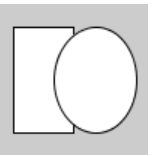
Recall, when we run this program, we see the following on the canvas:



Now, suppose that we would like to have the ellipse filled with yellow. To achieve this, we might try writing

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
fill(255, 255, 0);
```

However, when we run our program, we see the same result as before:



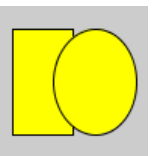
The ellipse that is drawn is not filled with yellow. This is because, given the order of the statements in our program, we set the fill color to yellow *after* the ellipse was already drawn. Thus, any *additional* shape that drawn *afterward* would be filled with yellow, but the fill color of any shape that has *already* been drawn will remain unchanged.

Once again, we see an important general principle at work here: when we wish to change a particular setting for drawing graphical elements, we must do this *before* we draw the graphical elements that we wish to be effected by this change.

In our current program, we need to move the current call to the `fill()` function so that it is *before* the call to the `ellipse()` function. If we position the call to the `fill()` function at the beginning of the program,

```
fill(255, 255, 0);
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

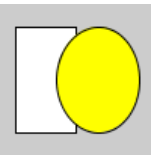
then both the rectangle and the ellipse are filled with yellow:



If we want to have only the ellipse filled with yellow, then we need to position the call to the `fill()` function *after* the call to the `rect()` function but *before* the call to the `ellipse()` function:

```
rect(10, 15, 40, 70);
fill(255, 255, 0);
ellipse(65, 50, 55, 70);
```

Now, when we run our program only the ellipse is filled with yellow, and the rectangle is filled with the default fill color, white:



It is important to remember the general principle that was previously mentioned:

> When we change a setting for the drawing of a graphical element, that change in the setting will *remain in effect* for any graphical elements that we draw *afterward*.

In the current program, any shapes that we draw after the ellipse will continue to be filled with yellow, unless we subsequently change the setting of the fill color using a call to the `fill()` function or to the `noFill()` function.

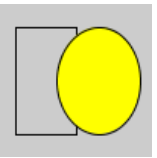
Processing also provides the **`noFill()`** function to *turn off* the filling of shapes. This function does not have any parameters. Thus, the general form of a call to this function is

```
noFill();
```

For example, let's add a call to the `noFill()` function at the beginning of our program:

```
noFill();
rect(10, 15, 40, 70);
fill(255, 255, 0);
ellipse(65, 50, 55, 70);
```

Notice that, as a result, the rectangle is now unfilled and, instead, the default color of the canvas now shows through on the inside of the rectangle:



In contrast, notice that the ellipse remains filled with yellow. This is because there is a call to the `fill()` function before we draw the ellipse:

```
noFill();
rect(10, 15, 40, 70);

fill(255, 255, 0);
ellipse(65, 50, 55, 70);
```

A call to the `fill()` function not only sets the fill color but also causes the filling of shapes to *resume*.

Changing the Stroke Color: The `stroke()` and `noStroke()` Functions

Notice that all of the points, line segments, and the edges of the shapes we have drawn in Processing have been black. This is because black is the default **stroke color**.

Processing provides the **`stroke()`** function that allows us to change the stroke color that will be used for any points, line segments, or shape edges that we draw *after calling this function*.

The basic form of a call to the **`stroke()`** function is

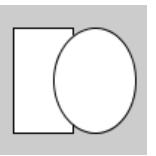
```
stroke(red, green, blue);
```

The parameters are

red	The amount of red (0 to 255)
green	The amount of green (0 to 255)
blue	The amount of blue (0 to 255)

Let's once again revise our `MoreDrawing` program to consisting of just the following two lines of code:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```



We'll now add a call to the `strokeWeight()` function at the beginning of our program:

```
strokeWeight(5);
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

As a result, when we run this program, the edges around our shapes are much more visible:

Next, let's place a call to the `stroke()` function at the beginning of our program that sets the stroke color to red:

```
stroke(255, 0, 0);
strokeWeight(5);
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

Thus, before we draw our shapes, we are now making changes to *two* stroke settings: the stroke **color** and the stroke **weight**. As a result, when we run our program, we see that each of our shapes is now drawn with a red stroke around it that has a thickness of 5 pixels:

Remember, the changing of a drawing property stays in effect for any shapes that are drawn *afterward*. Thus, in the current program, any stroking—that is, the drawing of points, line segments, or shape edges—that takes place *after* this call to the `stroke()` function will also be red in color and 5 pixels in thickness. This is why *both* the rectangle and the ellipse have been drawn with the specified stroke weight and stroke color.

Processing also provides the **`noStroke()`** function to *turn off* the drawing of edges around shapes. This function does not have any parameters. Thus, the general form of a call to this function is

```
noStroke();
```

To illustrate, let's change the call to the `stroke()` function at the beginning of our program to a call to the `noStroke()` function:

```
noStroke();
strokeWeight(5);
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

As a result, when we run our program, neither the rectangle nor the ellipse has an edge drawn around it:

Once the `noStroke()` function is called, no stroking will take place. There will be no drawing of points, line segments, or shape edges until there is a call to the `stroke()` function. For example, if we make the following change to our program,

```
noStroke();
strokeWeight(5);
rect(10, 15, 40, 70);
stroke(255, 0, 0);
ellipse(65, 50, 55, 70);
```

then the rectangle still does not have a stroke around it, but there is once again a red stroke around the ellipse because it is drawn *after* the call to the `stroke()` function:

Inline Comments

Processing gives us the option of putting **comments** in our code. One common use of comments is to clarify the purpose of one or more lines of our code, not only for others who might read our code but also for ourselves as we work on a program.

One form of commenting in Processing with is **inline** comments. If we type two forward slashes (//), Processing will *ignore* anything we write from the point where the slashes are inserted up to the end of that line of code.

For example, to clarify that the call to the `stroke()` function in our current program will turn on red stroking, we can add the following comment:

```
noStroke();
strokeWeight(5);
rect(10, 15, 40, 70);
stroke(255, 0, 0); // red stroke
ellipse(65, 50, 55, 70);
```

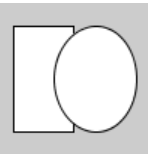
Notice that this does not cause an error, because Processing ignores everything we have written from the two slashes to the end of that line of code.

Grayscale

Let's once again edit our `MoreDrawing` program so that it consists of just the following two statements:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

When we run this program, we once again see the following drawn on the canvas:

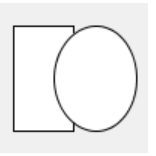


Our canvas color is currently the default shade of gray. As it turns out, the RGB values for this particular shade of gray are **204, 204, 204**.

Suppose that we wish to set our background to a lighter shade of gray instead. For example, we could do so by inserting the following call to the `background()`function at the beginning of our program:

```
background(240, 240, 240);
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

When we run this program, we now see a lighter shade of gray for the canvas:



Notice that, once again, all three of the RGB values for this shade of gray are the same: `240`, `240`, `240`.

Actually, like the RGB numbers for black—0, 0, 0—and the RGB numbers for white—255, 255, 255—*all three* of the RGB numbers for a shade of gray are always the *same*. Whenever red, green, and blue light are mixed in an equal amount that is greater than 0 and less than 255, some shade of gray light is produced.

Another way of visualizing this option is in terms of a **grayscale**, running from black to white through many shades of gray. In Processing, any color on this grayscale is described by a set of three identical RGB numbers, from 0, 0, 0 (black) to 255, 255, 255 (white):

Because all three of the RGB numbers for any color on this grayscale are always the same, Processing also permits us to refer to black, white, or a particular shade of gray using just *one* of the three identical RGB numbers. Thus, in our current example, instead of our current statement

```
background(240, 240, 240);
```

we actually have the option of simply writing

```
background(240);
```

Thus, we can also think of the color numbers of this grayscale as a range of single numbers from 0 to 255.

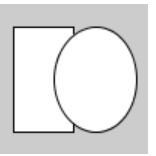
However, for consistency, in this book we will always use a set of three RGB numbers to describe any color, even when that color is black, white, or a shade of gray.

Transparency

Let's once again return our `MoreDrawing` program to the starting point that consists of just the following two lines of code:

```
rect(10, 15, 40, 70);
ellipse(65, 50, 55, 70);
```

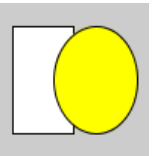
When we run this program, we once again see the following drawn on the canvas:



Let's again set the fill color of the ellipse to yellow using the following call to the `fill()` function:

```
rect(10, 15, 40, 70);
fill(255, 255, 0);
ellipse(65, 50, 55, 70);
```

When we run this program, the fill color of the ellipse is indeed set to yellow:



As we have seen, an RGB color description consists of three numbers, each of which is from 0 to 255. This describes a color that is totally **opaque**, not at all transparent. Thus, in our current example, the portion of the rectangle that the ellipse overlaps is completely invisible.

However, Processing also allows us the option of adding a *fourth* number to describe the **transparency** of the selected color. This fourth number is also in the range from 0 to 255, where 0 is a color that is totally transparent and 255 is a color that is totally opaque (the default). This number is sometimes known as the color's **opacity**, a term that refers to how *opaque* the color is. It is also known as the **alpha** value, and accordingly, this use of four numbers in this manner is sometimes known as the **RGBA** color model.

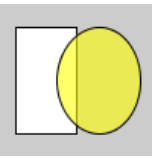
In our current example, the call to the `fill()` function that sets the fill color of our ellipse is

```
fill(255, 255, 0);
```

If we would prefer to have our ellipse filled instead with a *semitransparent* yellow, we simply add a *fourth* argument to the call to the `fill()` function. This fourth argument can be any value between 0 (totally transparent) and 255 (totally opaque). For example, we can write

```
rect(10, 15, 40, 70);
fill(255, 255, 0, 150);
ellipse(65, 50, 55, 70);
```

Now, when we run our program, the ellipse is filled with a semitransparent yellow, and as a result, the portion of the rectangle that the ellipse overlaps is now partially visible:



Such transparency can also be used with the `stroke()` function to create a point, line segment, or shape edge that is semitransparent.

However, using the four-number RGBA model currently *has no effect* in Processing if it is used with the `background()` function.

Summary

In this section, we learned about a number of additional functions that can be used for drawing graphic elements in Processing.

`strokeWeight(pixels);`

Parameters

pixels	The desired stroke weight, specified in pixels

`background(red, green, blue);`

Parameters

red	The amount of red
green	The amount of green
blue	The amount of blue

`fill(red, green, blue);`

Parameters

red	The amount of red
green	The amount of green
blue	The amount of blue

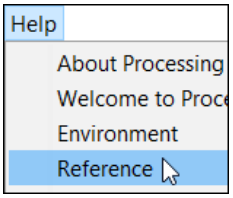
`noFill();`
Parameters: None

`stroke(red, green, blue);`
Parameters

red	The amount of red
green	The amount of green
blue	The amount of blue

`noStroke();`
Parameters: None

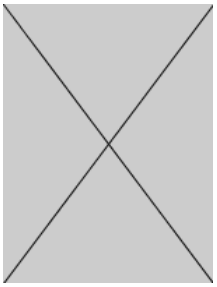
Additional information about all these functions can also be found in the **Reference** that is built into Processing and can be reached via the **Help** menu:



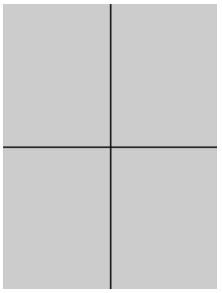
Processing has much more capability for drawing than was covered in this chapter. However, you are now well prepared to understand the graphical examples in the chapters ahead. You are also now in a great position to start having fun experimenting with drawing in Processing on your own.

Exercises

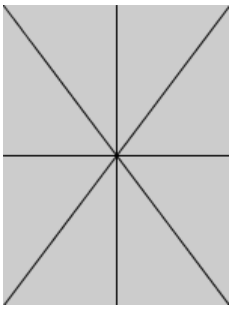
1) Write code that will draw the following on a 150-pixel by 200-pixel canvas:



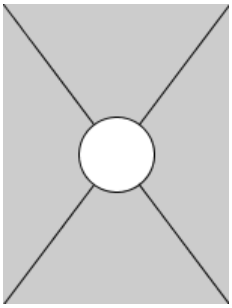
2) Write code that will draw the following on a 150-pixel by 200-pixel canvas:



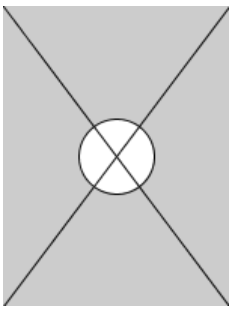
3) Write code that will draw the following on a 150-pixel by 200-pixel canvas:



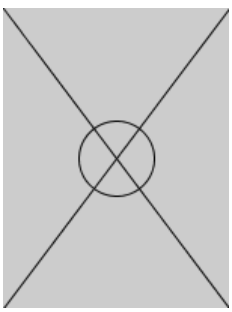
4) Write code that will draw the following on a 150-pixel by 200-pixel canvas:



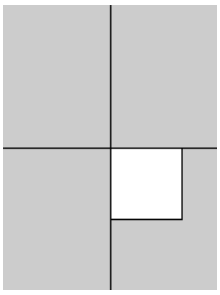
5) Write code that will draw the following on a 150-pixel by 200-pixel canvas:



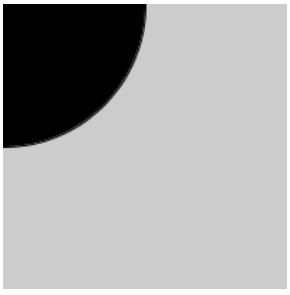
6) Write code that will draw the following on a 150-pixel by 200-pixel canvas:



7) Write code that will draw the following on a 150-pixel by 200-pixel canvas:

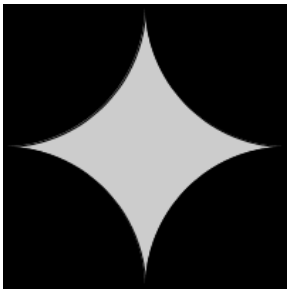


8) Using the `ellipse()` or `arc()` function, write code that will draw the following on a 200-pixel by 200-pixel canvas:

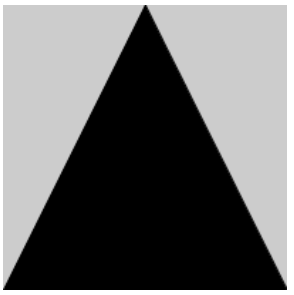


9) Using the `ellipse()` or `arc()` function, write code that will draw the following on a 200-pixel by 200-pixel canvas:

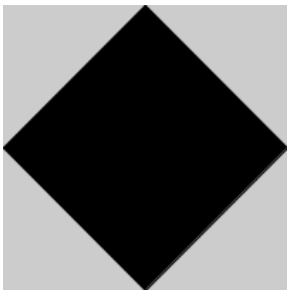
10) Using the `ellipse()` or `arc()` function, write code that will draw the following on a 200-pixel by 200-pixel canvas:



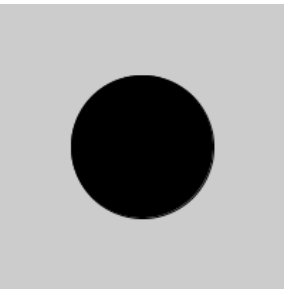
11) Write code that will draw the following on a 200-pixel by 200-pixel canvas:



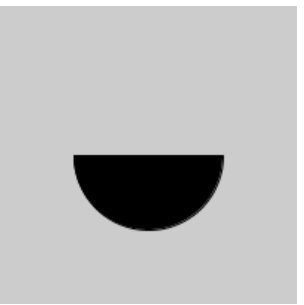
12) Use the `quad()` function to draw a diamond shape like the following on a 200-pixel by 200-pixel canvas:



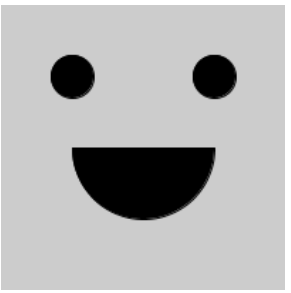
13) Write code that will draw the following on a 200-pixel by 200-pixel canvas:



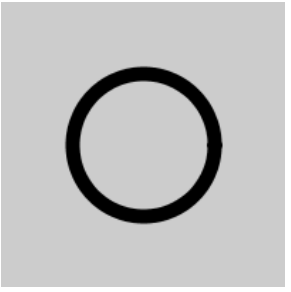
14) Use the `arc()` function to draw the following on a 200-pixel by 200-pixel canvas:



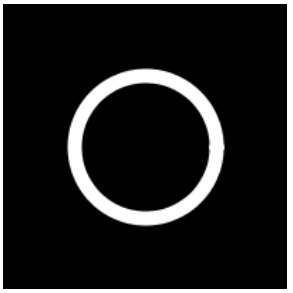
15) Write code that will draw the following on a 200-pixel by 200-pixel canvas:



16) Write code that will draw the following on a 200-pixel by 200-pixel canvas:



17) Write code that will draw the following on a 200-pixel by 200-pixel canvas:

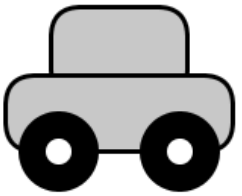


18) Draw a simple house using Processing code. (Tip: Draw on graph paper first.)

Example

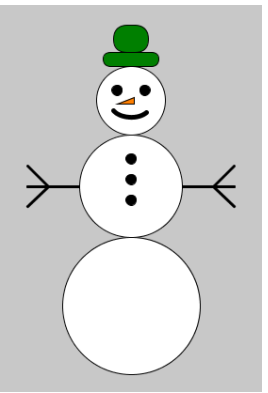
19) Draw a toy car using Processing code.

Example



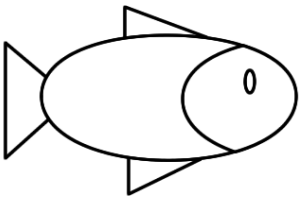
20) Draw a snowman using Processing code.

Example



21) Draw an animal using Processing code.

Example



22) Create your initials using Processing code.

23) Pick a national flag with a simple design and try to recreate it with Processing code.

24) Try to recreate the Olympic flag with Processing code.

25) Create a drawing in the style of the painter Piet Mondrian using Processing code.