Introduction

"I am two with nature."
—Woody Allen

Here we are: the beginning. Well, almost the beginning. This introduction is here to just get our feet wet. If it's been a while since you've done any programming in Processing (or any math, for that matter), this will get your mind back into computational thinking before we head into some of the more difficult and complex material.

In Chapter1, we're going to talk about the concept of a vector and how it will serve as the building block for simulating motion throughout this book. But before we take that step, let's think about what it means for something to even move around the screen. Let's begin with one of the best-known and simplest simulations of motion—the Random Walk.

I.1 Random Walks

Imagine you are standing in the middle of a balance beam. Every ten seconds, you flip a coin. Heads, take a step forward. Tails, take a step backwards. This is a random walk—a path that is defined as a series of random steps. Stepping off that balance beam and onto the floor, you could perform a random walk by flipping that same coin twice with the following results:

Flip 1	Flip 2	Result
Heads	Heads	Step forward.
Heads	Tails	Step right.
Tails	Heads	Step left.
Tails	Tails	Step backward.

Yes, this may seem like a particularly unsophisticated algorithm. Nevertheless, random walks can be used to model phenomena that occur in the real world, from the movements of molecules in a gas to the behavior of a gambler spending a day at the casino. In our case, we begin the random walk keeping three things in mind regarding this book with three goals in mind.

1) We need to review a programming concept central to this book—object-oriented programming. The random walker will serve as a template for how we will use object-oriented design to make things that move around a Processing window.

2) the random walk instigates the two questions that we will ask over and over again throughout this book: "How do we define the rules that govern the behavior of our objects?" and then "How do we implement these rules in Processing?"
3) Throughout the book, we'll periodically need a basic understanding of randomness, probability, and Perlin noise. The random walk will allow us to demonstrate a few key points that will come in handy later.

I.2 The Random Walker Class

Let's review a bit of object-oriented programming ("OOP") first by building a "Walker" object. This will be only a cursory review. If you have never worked with OOP before, you may want something more comprehensive. I'd suggest stopping here and reviewing the basics on the Processing web site before continuing: http://processing.org/learning/objects/.

An ***object*** in Processing is an entity that has both data and functionality. We are looking to design a Walker object that both keeps track of its data (where it exists on the screen) and has the capability to perform certain actions (such as draw itself or take a step).

A ***class*** is the template for building actual instances of objects. Think of a class as the cookie cutter where the objects are the cookies themselves. Let's begin by defining this template—what it means to be a Walker object.

The Walker only needs two pieces of data—a number for its x-location and one for its y-location.

```
class Walker {          $$ Objects have data
  int x;
  int y;
```

Every class must have a constructor, a special function that is called when the object is first created. You can think of it as the object's ***setup()***. There, we'll initialize the Walker's starting location (in this case, the center of the window).

```
  Walker() {            $$ Objects have a constructor where they are initialized
    x = width/2;
    y = height/2;
  }
```

Finally, in addition to data, classes can be defined with functionality. In this example, a Walker has two functions. We first write a function to display itself (as a white dot).

```
  void display() {            $$ Objects have functions
    stroke(255);
    point(x,y);
  }
```

The second function directs the object to take a step. Now, this is where things get a bit more interesting. Remember that floor on which we were taking random steps? Well, now we can use a Processing window in that same capacity. There are four possible steps—a step to the right can be simulated by incrementing x (x++); to the left by decrementing x (x--); a step forward by going down a pixel (y++) and a step backwards as up a pixel (y--). How do we pick from these four choices? Earlier we stated that we could flip two coins. In Processing, however, when we want to randomly choose from a list of options, we can pick a random number using ***random()***.

```
 void step() {

   int choice = int(random(4));    $$ 0, 1, 2, or 3
```

The above line of code picks a random floating point number between 0 and 4 and converts it to an integer, resulting in 0, 1, 2, or 3. Technically speaking, the highest number will never be 4.0, but rather 3.999999999 (with as many 9's as there are decimal places); since the conversion process to an integer lops off the decimal place, the highest int we can get is 3. Next, we take the appropriate step (left, right, up, or down) depending on which random number was picked.

```
   if (choice == 0) {              $$ The random "choice" determines our step
     x++;
   } else if (choice == 1) {
     x--;
   } else if (choice == 2) {
     y++;
   } else {
     y--;
   }
 }
}
```

Now that we've written the the template for making a Walker object, it's time to make an actual Walker object in the main part of our sketch—***setup()*** and ***draw()***. Assuming we are looking to model a single random walk, we declare one global variable of type Walker.

```
Walker w;              $$ A Walker object
```

Then we create the object in ***setup()*** by calling the constructor with the ***new*** operator.

Example: Traditional Random Walk

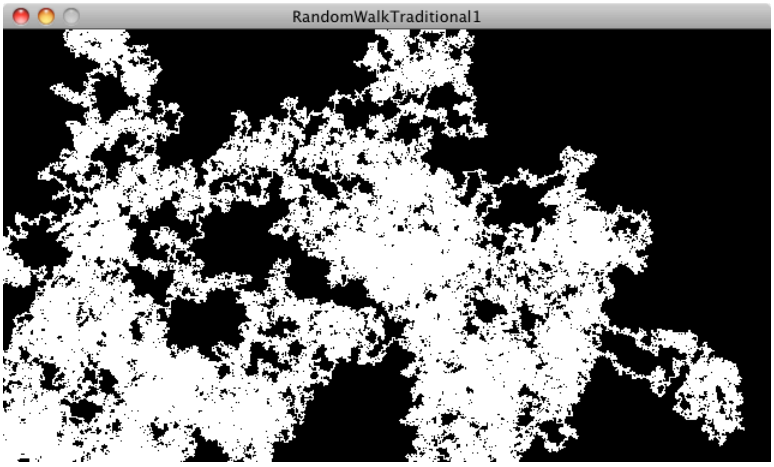
```
void setup() {
  size(640,360);
  w = new Walker();    $$ Create the Walker
  background(0);
}
```

Finally, during each cycle through ***draw(),*** we ask the Walker to take a step and draw a dot.

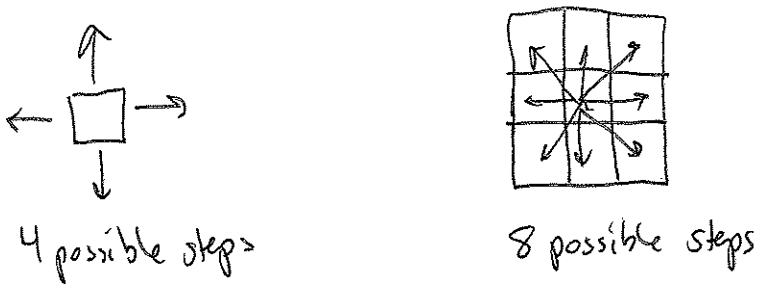
```
void draw() {
  w.step();            $$ Call functions on the Walker
  w.display();
```

```
}
```

Since we only draw the background once in ***setup()*** (rather than clearing it continually each time through ***draw()***), we see the trail of the random walk in our Processing window.



There are a couple improvements we could make with the random walker. For one, this walker's step choices are limited to four—up, down, left, and right. But any given pixel in the window has eight possible neighbors, and a ninth possibility is to stay in the same place.



To implement a walker that can step to any neighboring pixel (or stay put) we could then pick a number between zero and eight (nine possible choices). However, a more efficient way to write the code would be to simply pick from three possible steps along the x-axis (-1, 0, or 1) and three possible steps along the y-axis.

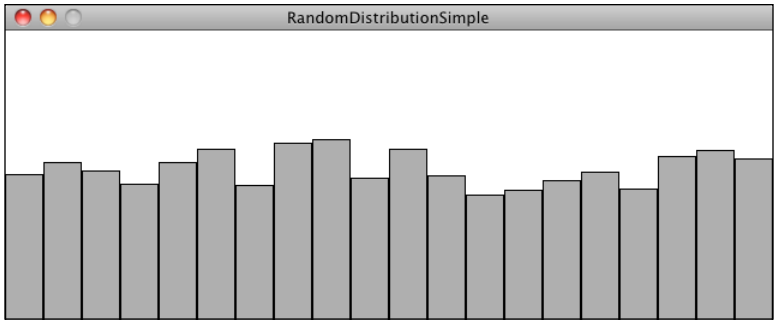
```
void step() {
  int stepx = int(random(3))-1;    $$ Yields -1, 0, or 1
  int stepy = int(random(3))-1;
  x += stepx;
  y += stepy;
}
```

Taking this a step further, we could use floating point numbers (i.e. decimal numbers) for x and y instead and move according to an arbitrary random value between -1 and 1.

```
void step() {
  float stepx = random(-1, 1);
  float stepy = random(-1, 1);
  x += stepx;
  y += stepy;
}
```

All of these variations on the "traditional" random walk have one thing in common: at any moment in time, the probability that the walker will take a step in a given direction is equal to the probability that the walker will take a step in any direction. In other words, if there are four possible steps, there is a one in four (or 25%) chance the walker will take any given step. With nine possible steps, it's a one in nine (or 11.1%) chance.

Conveniently, this is how the ***random()*** function works. Processing's random number generator (which operates behind the scenes) produces what is known as a "uniform" distribution of numbers. We can test this distribution with a Processing sketch that counts each time a random number is picked and graphs it as the height of a rectangle.



Example: Random Number Distribution

```
int[] randomCounts;           $$ An array to keep track of how often random numbers are picked


void setup() {
  size(640,240);
  randomCounts = new int[20];
}

void draw() {
  background(255);

  int index = int(random(randomCounts.length));  $$ Pick a random number and increase the count
  randomCounts[index]++;


  stroke(0);                  $$ Graphing the results
  fill(175);
  int w = width/randomCounts.length;
  for (int x = 0; x < randomCounts.length; x++) {
    rect(x*w,height-randomCounts[x],w-1,randomCounts[x]);
  }
}
```

The above screenshot shows the result of the sketch running for a few minutes. Notice how each bar of the graph differs in height. Our sample size (i.e. the number of random numbers we've picked) is rather small and there are some *random* discrepancies, where certain numbers are picked more often. Over time, with a good random number generator, this would even out.

Pseudo-Random Numbers
The random numbers we get from the ***random()*** *function are not truly random and are therefore known as "pseudo-random." They are the result of a mathematical function that simulates randomness. This function would yield a pattern over time, but that time period is so long that for us, it's just as good as pure randomness!*

I.3 Probability and Non-Uniform Distributions

Remember when you first started programming in Processing? Perhaps you wanted to draw a lot of circles on the screen. So you said to yourself: "Oh, I know. I'll draw all these circles at random locations, with random sizes, and random colors." In a computer graphics system, it's often easiest to seed a system with randomness. In this book, however, we're looking to build systems modeled on what we see in nature. Defaulting to randomness is not a particularly thoughtful solution to every design problem—in particular, the kind of problems that involve creating an organic or natural-looking design.

With a few tricks, we can change the way we use ***random()*** to produce "non-uniform" distributions of random numbers. This will come in handy throughout the book as we look at a number of different scenarios. When we examine genetic algorithms, for example, we'll need a methodology for performing "selection"—which members of our population should be selected to pass their DNA down to the next generation. Remember the concept of survival of the fittest? Let's say we have a population of monkeys evolving. Not every monkey will have a equal chance of reproducing. To simulate Darwinian evolution, we can't simply pick two random monkeys to be parents. We need the more "fit" ones to be more likely to be chosen. We need to define the "probability of the fittest." For example, perhaps a particularly fast and strong monkey has a 90% chance of procreating, while a weaker one has only a 10% chance.

Let's review the basic principles of probability, first looking at "Single Event Probability," i.e. the likelihood of something to occur.

Given a system with a certain number of possible outcomes, the probability of any given event occurring is the number of outcomes that qualify as that event divided by the total number of possible outcomes. The simplest example is a coin toss. There are a total of two possible outcomes (heads or tails). There is only one way to flip heads. Therefore, the probability of heads is one divided by two, i.e. 1/2 or 50%.

Consider a deck of fifty-two cards. The probability of drawing an ace from that deck is:

number of aces / number of cards = 4 / 52 = 0.077 = ~ 8%

The probability of drawing a diamond is:

number of diamonds / number of cards = 13 / 52 = 0.25 = 25%

We can also calculate the probability of multiple events occurring in sequence as the product of the individual probabilities of each event.

The probability of a coin coming up heads three times in a row is:

(1 / 2) * (1 / 2) * (1 / 2) = 1 / 8 (or 0.125).

In other words, a coin will land heads three times in a row one out of eight times (with each "time" being three tosses.)

Exercise: What is the probability of drawing two aces in a row from the deck of cards?

There are a few different techniques for using the ***random()*** function with probability in code. For example, if we fill an array with a selection of numbers (some repeated), we can randomly pick from that array and generate events based on what we select.

```
int[] stuff = new int[5];
stuff[0] = 1;                 $$ 1 is stored in the array twice to increase its likelihood
stuff[1] = 1;                 of being picked
stuff[2] = 2;
stuff[3] = 3;
stuff[4] = 3;
int index = int(random(stuff.length));  $$ Picking a random element from an array
if (stuff[index] == 1) {
```

If you run this code, there will be a 40% chance of printing the value 1, a 20% chance of printing 2, and a 40% chance of printing 3.

Another strategy is to ask for a random number (for simplicity, we consider random floating point values between 0 and 1) and allow an event to occur only if the random number we pick is within a certain range. For example:

```
float prob = 0.10;     $$ A probability of 10%
float r = random(1);   $$ A random floating point value between 0 and 1
if (r < prob) {        $$ If our random number is less than 0.1
   // DO SOMETHING!
}
```

This same technique can also be applied to multiple outcomes.

Outcome A — 60% | Outcome B — 10% | Outcome C — 30%

To implement this in code, we pick one random float and check where it falls.

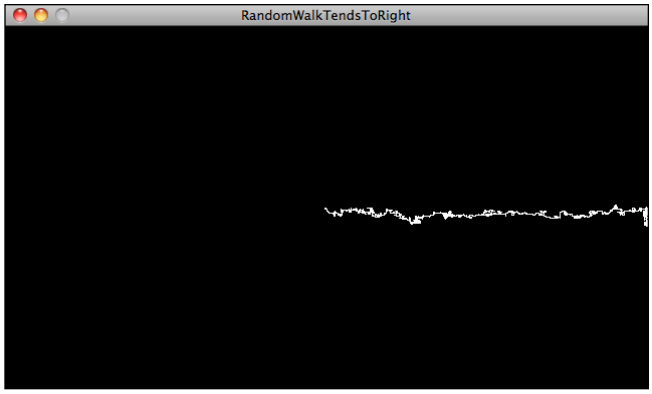
- *between 0.00 and 0.60 (60%) –> outcome A*
- *between 0.60 and 0.70 (10%) –> outcome B*

- *between 0.70 and 1.00 (30%) –> outcome C*

```
  float num = random(1);

  if (num < 0.6) {            $$ If random number is less than .6
    // Outcome A
  } else if (num < 0.7) {     $$ Between 0.6 or 0.7
    // Outcome B
  } else {                    $$ Greater than 0.7
    // Outcome C
  }
}
```

We could use the above methodology to create a random walker that tends to move to the right. Here is an example of a Walker with the following probabilities:

- *chance of moving up:* *20%*
- *chance of moving down:* *20%*
- *chance of moving left:* *20%*
- *chance of moving right:* *40%*




```
  Example: Walker that tends to move to the right
  void step() {

    float r = random(1);

    if (r < 0.4) {            $$ A 40% of moving to the right!
      x++;
    } else if (r < 0.6) {
      x--;
    } else if (r < 0.8) {
      y++;
    } else {
      y--;
    }
  }
```

Exercise: Create a random walker with dynamic probabilities. For example, can you give it a 50% chance of moving in the direction of the mouse?

I.4 A Normal Distribution of Random Numbers

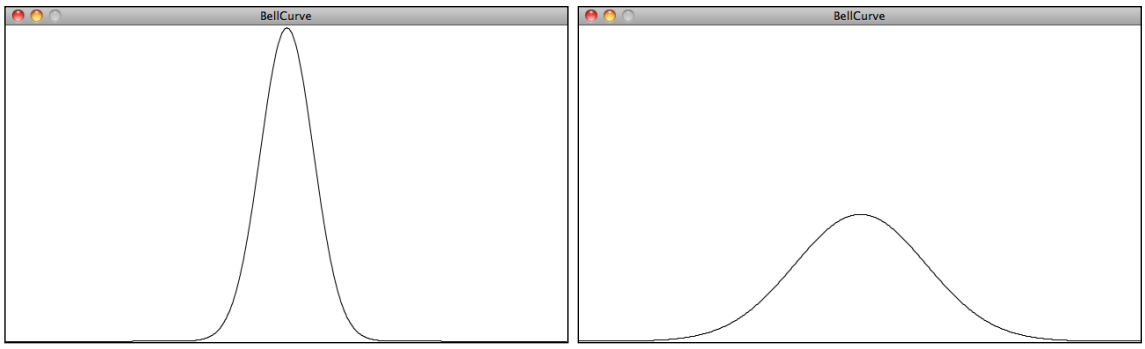
Let's go back to that population of simulated Processing monkeys. Your program generates a thousand "Monkey" objects with each monkey setting a value for height in the constructor—between 200 and 300 (this is a world of monkeys that have heights between 200 and 300 pixels).

```
float h = random(200,300);
```

Does this accurately depict the heights of real-world beings? Think of a crowded sidewalk in New York City. Pick a random person and it may appear that their height is random. Nevertheless, it's not the kind of random that the ***random()*** produces. People's heights are not uniformly distributed; there are a great deal more people of average height than there are very tall or very short ones. To simulate nature, we may want it to be more likely that our monkeys are of average height (250 pixels), yet allow them to still on occasion be very short or very tall.

A distribution of values that cluster around an average (referred to as the "mean") is known as a "normal" distribution. It is also called the Gaussian distribution (named for mathematician Carl Friedrich Gauss) or, if you are French, the Laplacian distribution (named for Pierre-Simon Laplace). Both mathematicians were working concurrently in the early nineteenth century on defining such a distribution.

When you graph the distribution, you get something that looks like the following, informally known as the bell curve.



The curve is generated by a mathematical function that defines the probability of any given value occurring as a function of the mean (often written as μ, the Greek letter *mu*) and standard deviation (σ, the Greek letter *sigma*).

The mean is pretty easy to understand. In the case of our height values between 200 and 300, we can intuitively have a sense of the mean (i.e. average) as 250. However, what if I were to say that the standard deviation is 3 or 15? The graphs above should give us a hint. The graph on the left shows us the distribution with a very low standard deviation, where the majority of the values cluster closely around the mean. The graph on the right shows us a higher standard deviation, where the values are more evenly spread out from the average.

The numbers work out as follows. Given a population, 68% of the members of that population will have values in the range of one standard deviation from the mean, 98% within two standard

deviations, 99.7% within three standard deviations. Given a standard deviation of five pixels, only 0.3% of the monkey heights will be less than 235 pixels (three standard deviations below the mean of 250) or greater than 265 pixels (three standard deviations above the mean of 250).

Calculating Mean and Standard Deviation

Consider a class of ten students who receive the following scores (out of 100) on a test:

85, 82, 88, 86, 85, 93, 98, 40, 73, 83

The mean is the average: 81.3

The standard deviation is calculated as the square root of the average of the squares of deviations aroundthe mean. In other words, take the difference from the mean for each person and square it (variance). Calculate the average of all these values and take the square root as our standard deviation.

Score	Difference from Mean	Variance
85	85-81.3 = 3.7	$(3.7)^2 = 13.69$
40	40-81.3 = -41.3	$(-41.3)^2 = 1705.69$
etc.		
	Average Variance:	254.23

The standard deviation is the square root of the average variance = 15.13

Luckily for us, to use a normal distribution of random numbers in a Processing sketch, we don't have to do any of these calculations ourselves. Instead, we can make use of a class known as Random, which we get for free as part of the default Java libraries imported into Processing (see: http://docs.oracle.com/javase/6/docs/api/java/util/Random.html for more information).

To use the Random class, we must first declare a variable of type Random and create the Random object in ***setup()***.

```
Random generator;                    $$ We use the variable name "generator" as what we
                                     have here can be thought of as a random number generator

void setup() {
  size(640,360);
  generator = new Random();
}
```

If we want to produce a random number with a normal (or Gaussian) distribution each time we run through ***draw()***, it's as easy as calling the function ***nextGaussian()***.

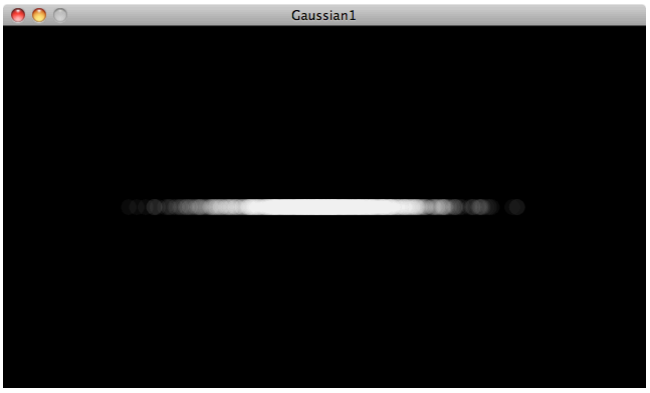
```
void draw() {
  float num = generator.nextGaussian();      $$ Asking for a Gaussian random number
}
```

Here's the thing. What are we supposed to do with this value? What if we wanted to use it to, for example, assign the x-position of a shape we draw on screen?

The ***nextGaussian()*** function returns a normal distribution of random numbers with the following parameters: ***a mean of zero*** and ***a standard deviation of one***. Let's say we want a mean of 360 (the center horizontal pixel in a window of width 640) and a standard deviation of 60 pixels. We can adjust the value to our parameters by multiplying it by the standard deviation and adding the mean.

Example: Gaussian Distribution

```
void draw() {
  float num = (float) generator.nextGaussian();      $$ Note nextGaussian() returns a double
  float sd = 60;
  float mean = 360;

  float x = sd*num + mean;            $$ Multiply by standard deviation and add the mean

  noStroke();
  fill(255,10);
  ellipse(x,180,16,16);
}
```



By drawing the ellipses on top of each other with some transparency, we can see the distribution visually. The brightest spot is near the center, where most of the values cluster, but every so often circles are drawn farther to the right or left of the center.

Exercise: Consider a simulation of paint splatter drawn as a collection of colored dots. Most of the paint clusters around a central location, but some dots do splatter out towards the edges. Can you use a normal distribution of random numbers to generate the locations of the dots? Can you also use a normal distribution of random numbers to generate a palette of color?

Exercise: A Gaussian random walk is defined as one in which the step size (how far you move in a given direction) is generated with a normal distribution. Implement this variation of our random walk.

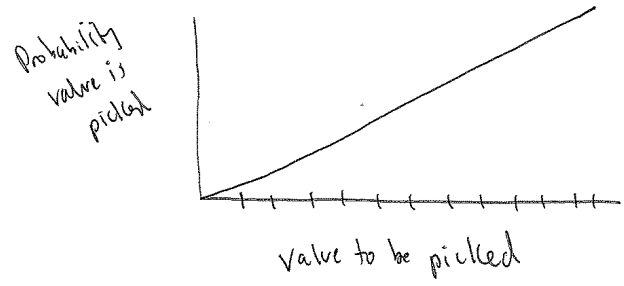
I.5 A Custom Distribution of Random Numbers

There will come a time in your life where you do not want a uniform distribution of random values or a Gaussian one. Let's imagine for a moment that you are a random walker in search of food. Moving randomly around a space seems like a reasonable strategy for finding something to eat. After all, you don't know where the food is, so you might as well search randomly until you find it. The problem, as you may have noticed, is that random walkers return to previously visited locations many times (this is known as "oversampling"). One strategy to avoid such a problem is to, every so often, take a very large step. This allows the walker to forage randomly around a specific location while periodically jumping very far away to reduce the amount of oversampling. This variation on the random walk (known as a Lévy flight) requires a custom set of probabilities. Though not an exact implementation of a Lévy flight, we could state the probability distribution as follows: the longer the step, the less likely it is to be picked; the shorter the step, the more likely.

Earlier in this prologue, we saw that we could generate custom probability distributions by filling an array with values (some duplicated so that they would be picked more frequently) or by testing the result of ***random()***. Certainly, we could implement a Levy flight by saying there is a 1% chance of the walker taking a large step.

```
float r = random(1);
if (r < 0.01) {               $$ A 1% chance of taking a large step
  xstep = random(-100,100);
  ystep - random(-100,100);
} else {
  xstep = random(-1,1);
  ystep - random(-1,1);
}
```

However, this reduces the probabilities to a fixed number of options. What if we wanted to make a more general rule—the higher a number, the more likely it is to be picked? 3.145 would be more likely to be picked than 3.144, even if that likelihood is just a tiny bit greater. In other words, if x is the random number, we could map the likelihood on the y-axis with y = x.



If we can figure out how to generate a distribution of random numbers according to the above graph, then we will be able to apply the same methodology to any curve for which we have a formula.

One solution is to pick two random numbers instead of one. The first random number is just that, a random number. The second one, however, is what we'll call a "qualifying random value." It

will tell us whether to use the first one or throw it away and pick another one. Numbers that have an easier time "qualifying" will be picked more often, and numbers that rarely qualify will be picked infrequently. Here are the steps (let's consider for now only random values between 0 and 1).

1. Pick a random number: R1
2. Compute a probability P that R1 should qualify. Let's try: P = R1.
3. Pick another random number: R2
4. If R2 is less than P, then we have found our number—R1!
5. If R2 is not less than P, go back to step 1 and start over.

Here we are saying that the likelihood that a random value will qualify is equal to the random number itself. Let's say we pick 0.1 for R1. This means that R1 will have a 10% chance of qualifying. If we pick 0.83 for R1 then it will have a 83% chance of qualifying. The higher the number, the greater the likelihood that we will actually use it.

Here is a function (named for the Monte Carlo method, which was named for the Monte Carlo casino) that implements the above algorithm, returning a random value between zero and one.

```
float montecarlo() {
  while (true) {       $$ We do this "forever" until we find a qualifying random value

    float r1 = random(1);      $$ Pick a random value
    float probability = r1;    $$ Assign a probability

    float r2 = random(1);      $$ Pick a second random value
    if (r2 < probability) {    $$ Does it qualify?  If so, we're done!
      return r1;
    }
  }
}
```

Exercise: Use a custom probability distribution to vary the size of a step taken by the random walker. The step size can be achieved by affecting the range of values picked. Can you map the probability exponentially—i.e. making the likelihood a value is picked equal to the value squared?

```
  float stepsize = random(-10,10);  $$ A uniform distribution of step sizes. Change this!

  float stepx = random(-stepsize,stepsize);
  float stepy = random(-stepsize,stepsize);

  x += stepx;
  y += stepy;
```

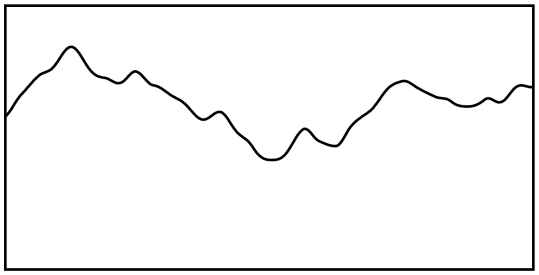
(Later we'll see how to do this more efficiently using vectors.)

I.6 Perlin Noise (A Smoother Approach)

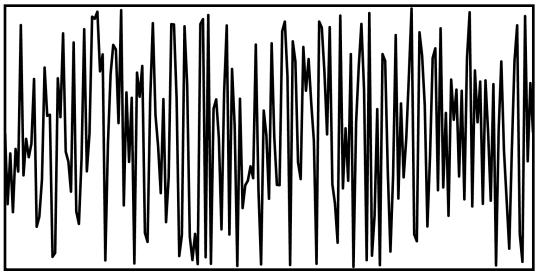
One of the qualities of a good random number generator is that the numbers produced have no relationship. If they exhibit no discernible pattern, they are considered *random*.

As we are beginning to see, a little bit of randomness can be a good thing when programming organic, life-like behaviors. However, randomness as the single guiding principle is not necessarily natural. An algorithm known as "Perlin noise," named for its inventor Ken Perlin, takes this concept into account. Perlin developed the noise function while working on the original *Tron* movie in the early 1980s. It was originally designed to create procedural textures for computer generated effects; in 1997 Perlin won an Academy Award in Technical Achievement for this work. Perlin noise can be used to generate a variety of interesting effects such as clouds, landscapes, and patterned textures like marble.

"Perlin noise" has a more organic quality because it produces a naturally ordered (i.e. "smooth") sequence of pseudo-random numbers. The graph on the left below shows Perlin noise over time (the x-axis represents time; note how the curve is smooth) while the graph on the right shows pure random numbers over time. (The code for generating these graphs is available with the accompanying book downloads.)



Perlin Noise



Random

Noise Detail

If you visit the *Processing.org* noise reference, you'll find that noise is calculated over several "octaves." You can change the number of octaves and their relative importance by calling the noiseDetail() function. This in turn changes how the noise function behaves . http://processing.org/reference/noiseDetail_.html

You can learn more about how noise works from Ken Perlin himself: http://www.noisemachine.com/talk1/

Processing has a built-in implementation of the Perlin noise algorithm with the function ***noise()***. The ***noise()*** function takes one, two, or three arguments (referring to the "space" in which noise is computed: one, two, or three dimensions.) Let's start by looking at one-dimensional noise.

Consider for a moment drawing a circle in our Processing window at a random x-location.

```
float x = random(0,width);     $$ A random x-location
ellipse(x,180,16,16);
```

Now, instead of a random xlocation, we want a Perlin noise x-location that is "smoother." You might think that all you need to do is replace ***random()*** with ***noise()***, i.e.

```
~~float x = noise(0,width);~~     $$ A noise x-location?
```

While conceptually this is exactly what we want to do—calculate an x-value that ranges between zero and the width according to Perlin noise—this is not the correct implementation. While the

arguments to the ***random()*** function specify a range of values between a minimum and a maximum, ***noise()*** does not work this way. Instead, the output range is fixed—it always returns a value between zero and one. We'll see in a moment that we can get around this easily with Processing's ***map()*** function, but first we must examine what exactly ***noise()*** expects us to pass in as an argument.

We can think of one-dimensional Perlin noise as a linear sequence of values over time. For example:

Time	Noise Value
0	0.365
1	0.363
2	0.363
3	0.364
4	0.366

Now, in order to access a particular noise value in Processing, we have to pass a specific moment in time to the ***noise()*** function. For example:

```
float n = noise(3);
```

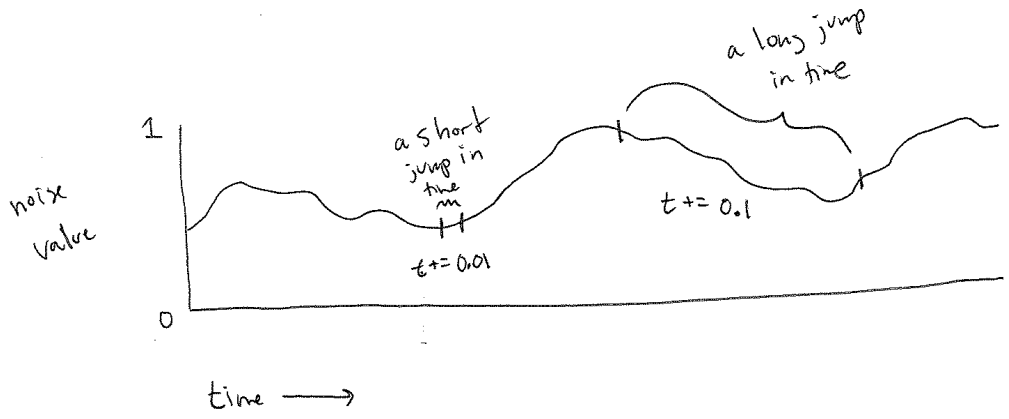
According to the above table, ***noise(3)*** will return 0.364 at time equals three. We could improve this by using a variable for "time" and asking for a noise value continuously in ***draw()***.

```
float t = 3;

void draw() {
  float n = noise(t);  $$ We need the noise value for a specific "moment in time"
  println(n);
}
```

The above code results in the same value printed over and over. This is because we are asking for the result of the ***noise()*** function at the same point in "time"—3—over and over. If we increment the "time" variable ***t***, however, we'll get a different result.

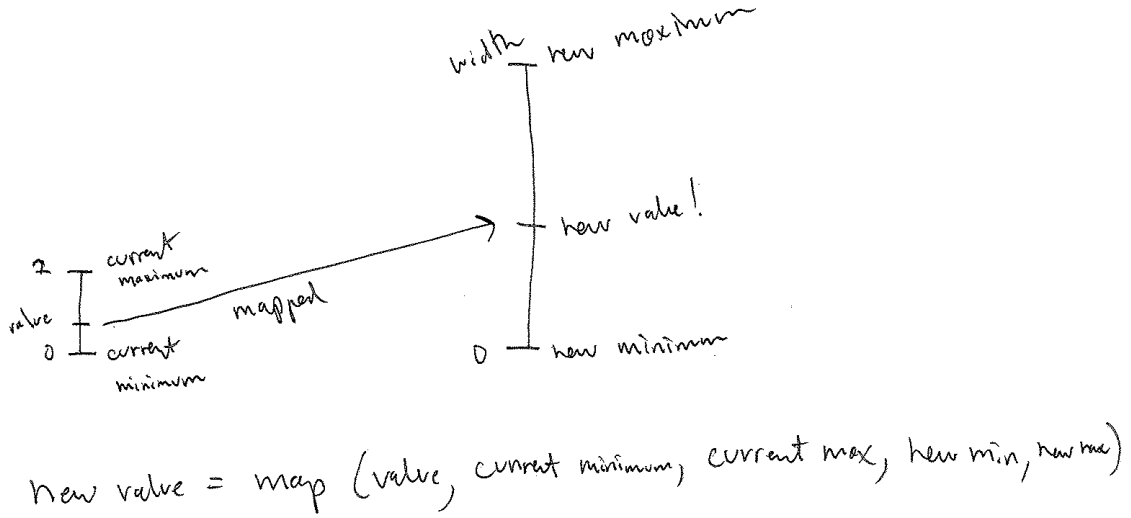
```
float t = 0;           $$ Typically we would start at time = 0, though this is arbitrary

void draw() {
  float n = noise(t);
  println(n);

  t += 0.01;           $$ Now, we move forward in time!
}
```

How quickly we increment "t" also affects the smoothness of the noise. If we make large jumps in time, then we are skipping ahead and the values will be more random.



Try running the code several times, incrementing t by 0.01, 0.02, 0.05, 0.1, 0.0001, and you will see different results.

Now we're ready to answer the question of what to do with the noise value. Once we have the value with a range between zero and one, it's up to us to map that range to what we want. The easiest way to do this is with Processing's ***map()*** function. The ***map()*** function takes five arguments. First up is the value we want to map, in this case ***n***. Then we have to give it the value's current range (minimum and maximum) followed by our desired range.



In this case, we know that noise has a range between zero and one, but we'd like to draw our circle with a range between zero and the window's width.

```
float t = 0;

void draw() {
  float n = noise(t);
  float x = map(n,0,1,0,width);       $$ Using map() to customize the range of Perlin noise
  ellipse(x,180,16,16);

  t += 0.01;
```

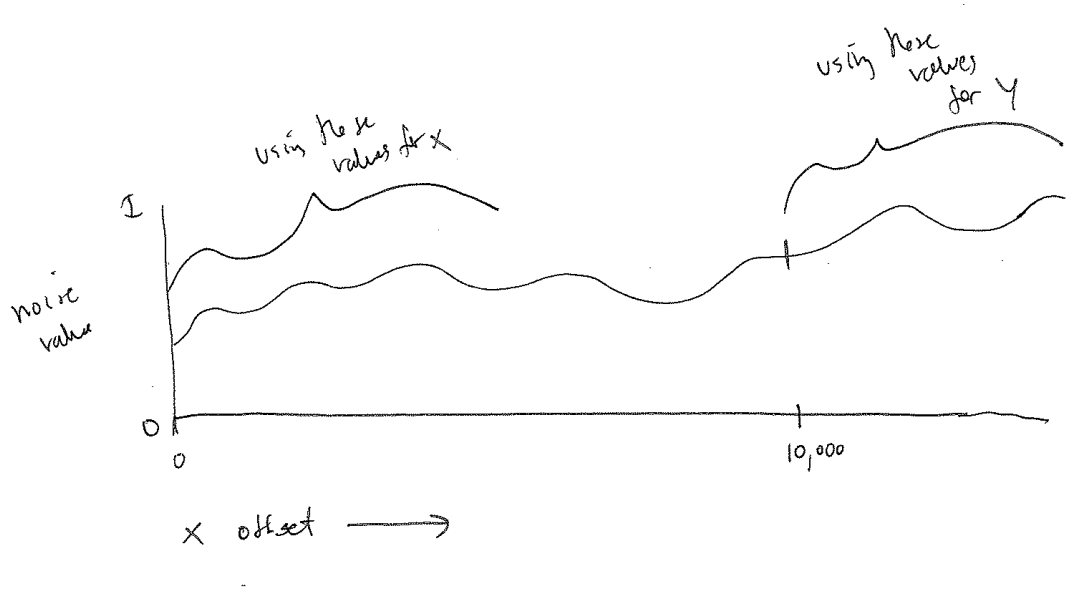
```
}
```

We can apply the exact same logic to our random walker, and assign both its x- and y-values according to Perlin noise.

Example: Noise Walker

```
class Walker {
  float x,y;

  float tx,ty;

  Walker() {
    tx = 0;
    ty = 10000;
  }

  void step() {
    x = map(noise(tx), 0, 1, 0, width);       $$ x- and y-location mapped from noise
    y = map(noise(ty), 0, 1, 0, height);

    tx += 0.01;                               $$ Move forward through "time"
    ty += 0.01;
  }
}
```

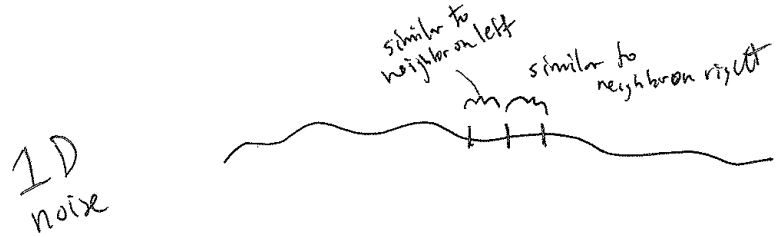
Notice how the above example requires an additional pair of variables: "tx" and "ty". This is because we need to keep track of two "time" variables, one for the x-location of the walker and one for the y. But there is something a bit odd about these variables. Why does tx start at zero and ty at 10,000? While these numbers are arbitrary choices, we have very specifically initialized our two time variables with different values. This is because the noise function is deterministic; it gives you the same result for a specific time t each and every time. If we asked for the the noise value at the same time t for both x and y, then x and y would always be equal, resulting in the walker only moving along a diagonal. Instead, we simply use two different parts of the noise space, starting at 0 for x and 10,000 for y so that x and y can appear to act independently of each other.



In truth, there is no true concept of "time" at play here. It's a useful metaphor to help us understand how the noise function works, but really what we have is space, rather than time. The graph above depicts a linear sequence of noise values in a one-dimensional space, and we can ask for a value at a specific x-location whenever we want. In examples, you will often see a variable named "xoff" to indicate the "x offset" along the noise graph rather than "t" for time (as noted in the diagram).

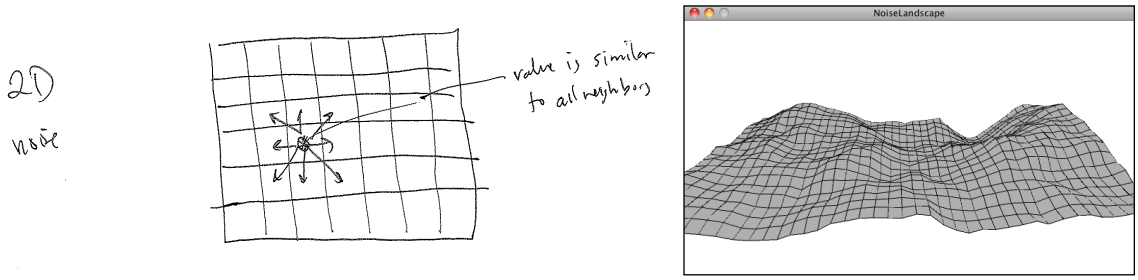
Exercise: In the above random walker, the result of the noise function is mapped directly to the walker's location. Create a random walker where you instead map the result of the noise() function to a walker's step size?

The reason why this idea of noise values living in a one-dimensional space is important is that it leads us right into a discussion of two-dimensional space. Let's think about this for a moment. With one-dimensional noise, we have a sequence of values in which any given value is similar to its neighbor. Because the value is in one dimension, it only has two neighbors: a value that comes before it (to the left on the graph) and one that comes after it (to the right).

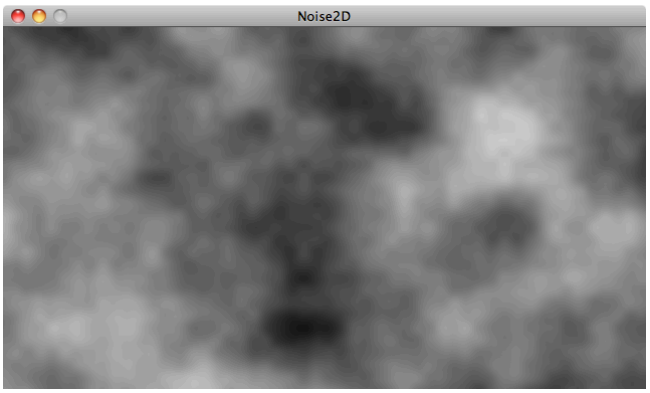


Two-dimensional noise works exactly the same way conceptually. The difference of course is that we aren't looking at values along a linear path, but values that are sitting on a grid. Think of

a piece of graph paper with numbers written into each cell. A given value will be similar to all of its neighbors: above, below, to the right, left, and along any diagonal.



If you were to visualize this graph paper with each value mapped to the brightness of a color, we would get something that looks like clouds. White sits next to light gray, which sits next to gray, which sits next to dark grey, which sits next to black, which sits next to dark gray, etc.



This is what noise was originally invented for. Tweak the parameters a bit, play with color, and the resulting image might look more like marble or wood or any other organic-looking texture.

Let's take a quick look at how you implement two-dimensional noise in Processing. If you wanted to color every pixel of a window randomly, you would need a nested loop, one that accessed each pixel and picked a random brightness.

```
loadPixels();
for (int x = 0; x < width; x++) {
  for (int y = 0; y < height; y++) {
    float bright = random(255);          $$ A random brightness!
    pixels[x+y*width] = color(bright);
  }
}
updatePixels();
```

To color each pixel according to the ***noise()*** function, we'll do exactly the same thing, only instead of calling ***random()*** we'll call ***noise()***.

```
    float bright = map(noise(x,y),0,1,0,255);
```

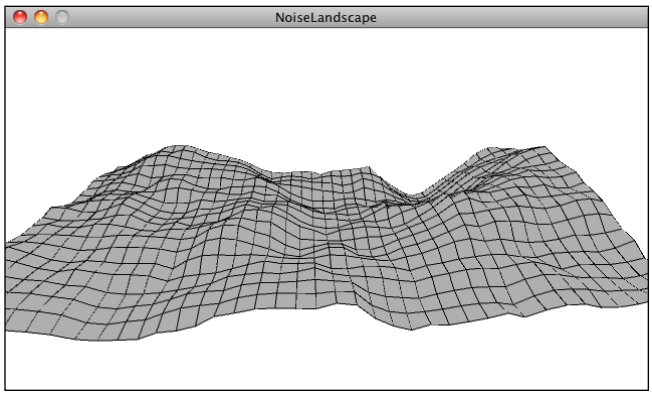
This is a nice start conceptually—it gives you a noise value for every xy location in our two-dimensional space. The problem is that this won't have the cloudy quality we want. Jumping from pixel 200 to pixel 201 is too large of a jump through noise. Remember, when we worked with one-dimensional noise, we incremented our "time" variable by 0.01 each frame, not by 1! A pretty good solution to this problem is to just use different variables for the arguments to noise. For example, we could increment a variable called "xoff" each time we move horizontally, and a "yoff" variable each time we move vertically through the nested loops.

```
float xoff = 0.0;        $$ Start xoff at 0

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

  float yoff = 0.0;      $$ For every xoff, start yoff at 0

  for (int y = 0; y < height; y++) {
    float bright = map(noise(xoff,yoff),0,1,0,255);       $$ Use xoff and yoff for noise()
    pixels[x+y*width] = color(bright);                    $$ Use x and y for pixel location

    yoff += 0.01;   $$ Increment xoff
  }
  xoff += 0.01;     $$ Increment xoff
}
```

Exercise: Play with color, noiseDetail(), and the rate at which xoff and yoff are incremented to achieve different visual effects.

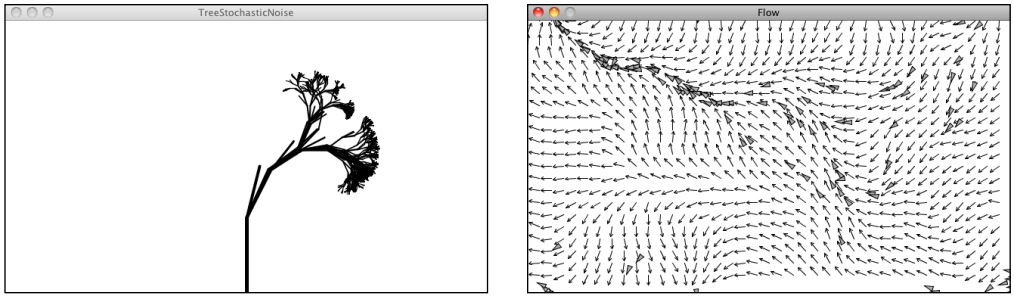
Exercise: Add a third argument to noise that increments once per cycle through ***draw()*** *to animate the two-dimensional noise.*

Exercise: Use the noise values as the heights of a landscape. See the screenshot below as a reference.



We've examined several traditional uses of Perlin noise in this section. With one-dimensional noise, we used smooth values to assign the location of an object to give the appearance of wandering. With two-dimensional noise, we created a cloudy pattern with smoothed values on a plane of pixels. It's important to remember, however, that Perlin noise values are just that—values. They aren't inherently tied to pixel locations or color. Any example in this book that has a variable could be controlled via Perlin noise. When we model a wind force, the strength of that

force could be controlled by Perlin noise. When we design a fractal tree pattern, the angles between the branches could be controlled by Perlin noise. When we develop a flow field simulation, the speed and direction of objects moving along a grid could be controlled by Perlin noise.



Tree with Perlin noise Flow field with Perlin noise

I.7 Onward

We began this chapter by talking about how randomness can be a crutch. In many ways, it's the most obvious answer to the kinds of questions we ask continuously—how should this object move? What color should it be? This obvious answer, however, can also be a lazy one.

As we finish off this prologue, it's also worth noting that we could just as easily fall into the trap of using Perlin noise as a crutch. How should this object move? Perlin noise! What color should it be? Perlin noise! How fast should it grow? Perlin noise!

The point of all of this is not to say that you should or should not use randomness. Or that you should or should not use Perlin noise. The point is that the rules of your system are defined by you and the larger your toolbox, the more choices you'll have as you implement those rules. The goal of this book is to fill your toolbox. If all you know is random, then your design thinking is limited. Sure, Perlin noise helps, but you'll need more. A lot more.

I think we're ready to begin.