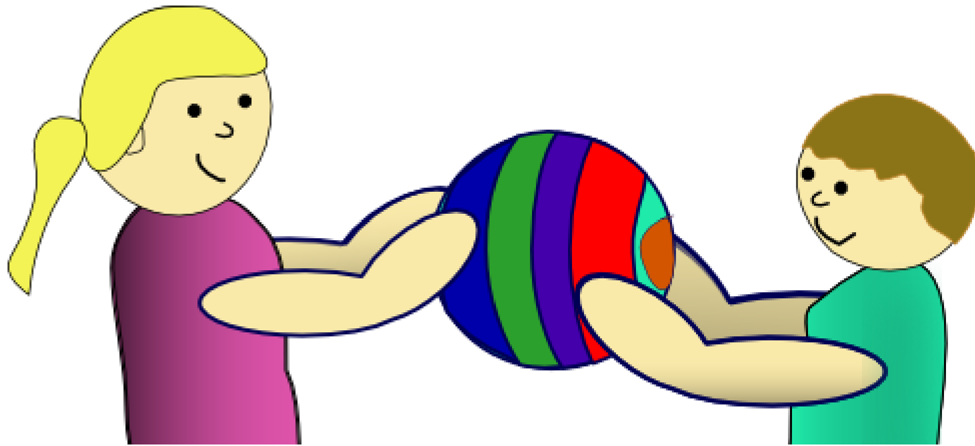
Chapter 8

Shared Data

What is in This Chapter ?

This chapter discusses the notion of **sharing data** in your programs. It explains situations in which sharing of data can be useful to simplify your program and be memory efficient. The chapter also explains how sharing of data is sometimes necessary in order for your program to run and then gives an example showing potentially unpleasant consequences of not being careful when dealing with shared data. Lastly, a large **graph editor** example is given that incorporates shared data and brings together many of the concepts that you have learned throughout this course.



8.1 Sharing Data Can Be Useful

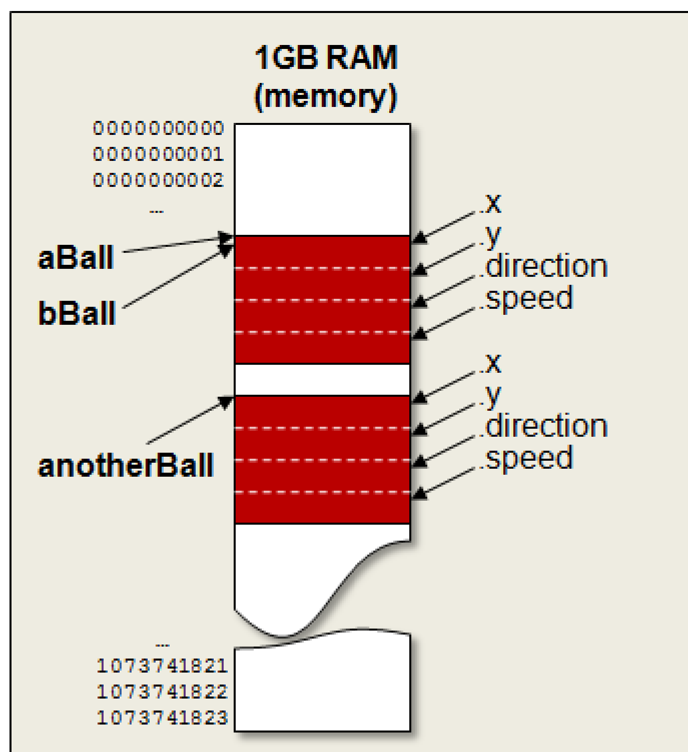
Recall in our discussion of variables that a variable is ***bound to*** *(i.e., attached to)* a value when we assign something to it using the **=** operator.

```
x = 100;
name = "Bob";
```

When we create our own data structures (i.e., objects) we need to remember that the data that makes up the object (i.e., the object's attributes) is stored in the computer's memory. The object itself is simply just a reference (i.e., a pointer) to the location in memory where the object's attributes are actually being stored. Each object created is a unique reference (i.e., memory location in the computer's memory).

Consider creating two ball objects as follows:

```
Ball    aBall;
Ball    anotherBall;

aBall = new Ball();
anotherBall = new Ball();
```



Since each ball is stored in a different memory location, each has its own set of unique attribute values. That is, each ball has its own (**x**,**y**) location, **direction** and **speed**.

Sometimes, however, we can end up in a situation in which two balls share the same memory location. Consider adding another ball variable as follows:

```
Ball    bBall;

bBall = aBall;
```

Now, **bBall** and **aBall** are actually references to the same object ... that is ... they are pointing to the same memory location and therefore share the same attributes.

Having two objects share the same memory can be advantageous since they can share the same information together, using up less memory space. For example, consider a "family car" in which many members of the same family drive the exact same vehicle, thereby reducing the need to buy another car and saving space in the laneway.

Of course, sharing does have some disadvantages. For example, if someone borrows the car and brings it back with no gasoline, then it affects the next person who will use the car. Also, the seat and mirrors may all need to be adjusted for the next driver. Worse yet, if one person in the family crashes the car, then the car is ruined for everyone.

When programming, it is important to understand when data structures are being shared so that efficient programs can be written, while ensuring that the data from one object does not interfere with the data from other objects unexpectedly.

Example:

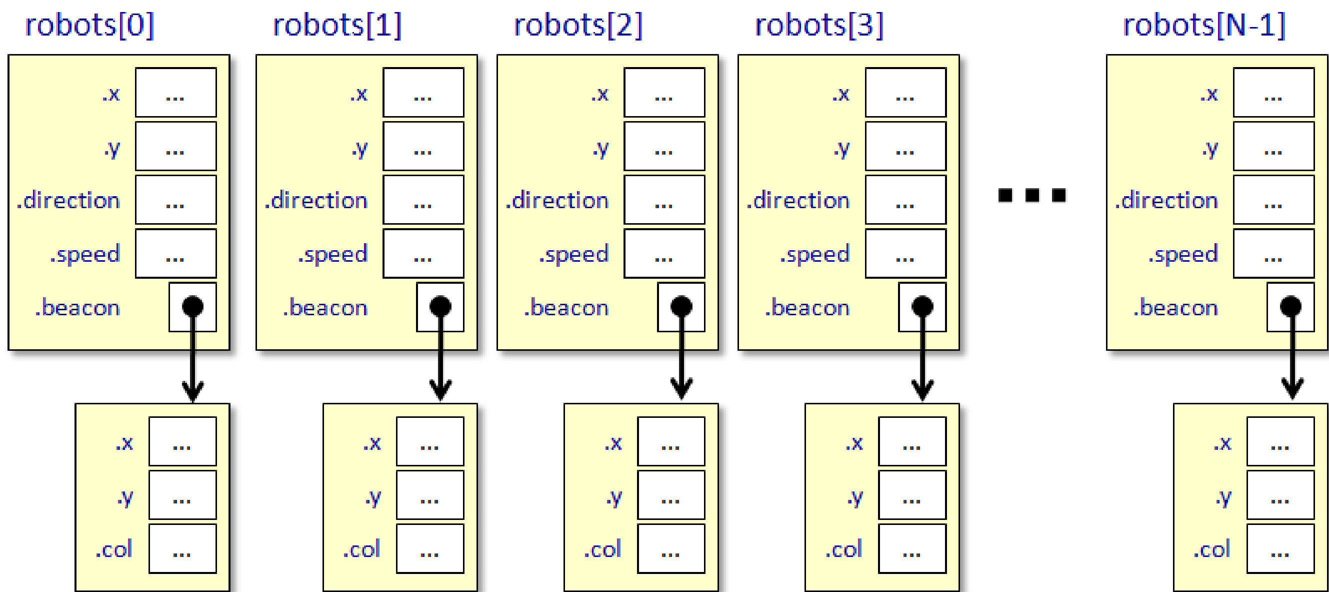
Consider simulating a swarm of insect-like robots that are attracted towards a beacon such as a light source. We can define a beacon as having a particular (x,y) location on the window and perhaps a randomly-chosen color:

```
class Beacon {
  float         x, y;
  color         col;

  Beacon(float bx, float by) {
    x = bx;
    y = by;
    col = color(55+random(200),55+random(200),55+random(200));
  }
}
```

Now to simulate the robots, let us define robot's as having an (**x**, **y**) location, a **direction** and a **speed**. We will also assign a **Beacon** to each robot as some particular location to head towards. The definition is very similar to the **Ball** data structure that we defined previously:

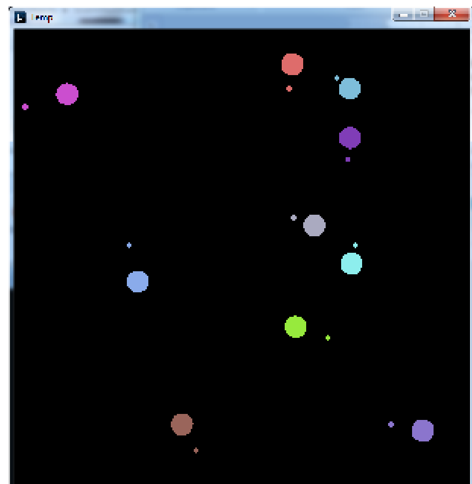
```
class Robot {
  float     x, y;               // location of the robot at any time
  float     direction;          // direction of the robot at any time
  float     speed;              // the robot's speed
  Beacon    beacon;             // the beacon to head towards

  Robot(float rx, float ry) {
    x = rx;
    y = ry;
    direction = random(TWO_PI);      // a random direction
    speed = 3 + random(4);           // a random speed from 3 to 6
    beacon = null;                   // not set yet
  }
}
```

Notice that each robot now stores their own **Beacon** object (i.e., their own place to head towards). We can draw the data structures as follows ... notice how each robot has its own **Beacon** object, allowing them all to head in different directions:



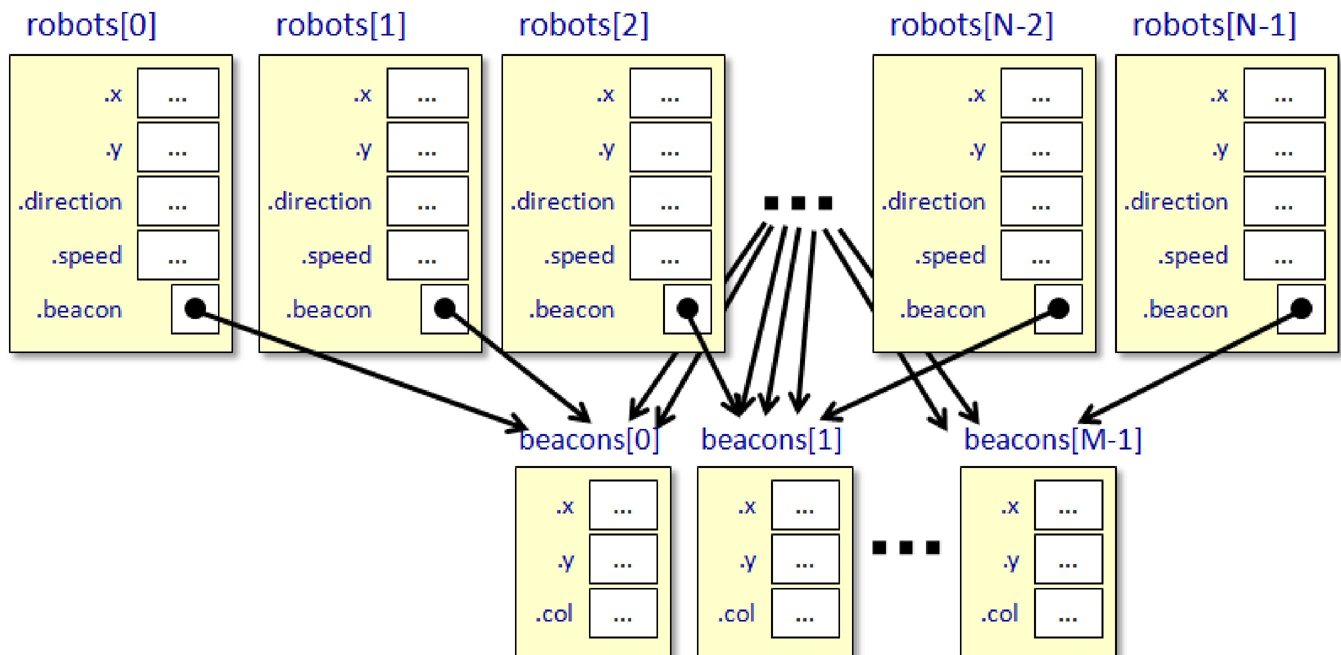
Here is the additional code required to create and display the robots and their beacons:

```
Robot[]       robots;                 // a bunch of robots
final int     ROBOT_RADIUS = 2;       // a robot's radius (in pixels)
final int     BEACON_RADIUS = 15;     // a beacon's radius (in pixels)

void setup() {
  size(600,600);

  // Make some robots with unique beacons
  robots = new Robot[10];
  for (int i=0; i<robots.length; i++) {
    robots[i] = new Robot(width/2, height/2);
    robots[i].beacon = new Beacon(random(width), random(height));
  }
}

void draw() {
  background(0,0,0);
  for (int i=0; i<robots.length; i++) {    // draw all robots and their beacons
    draw(robots[i]);
    draw(robots[i].beacon);
    move(robots[i]);   // explained later
  }
}

void draw(Robot r) {
  fill(r.beacon.col);  // use beacon's color
  stroke(0,0,0);
  ellipse(r.x, r.y, 2*ROBOT_RADIUS,2*ROBOT_RADIUS);
}

void draw(Beacon b) {
  fill(b.col);
  stroke(0,0,0);
  ellipse(b.x, b.y, 2*BEACON_RADIUS,2*BEACON_RADIUS);
}
```



Notice how each robot is colored to match its beacon location's color. The **move()** procedure will be explained later. For now, assume though that it steers the robot towards its beacon location.

In the code above, each robot has its own unique beacon to head towards. Now we will adjust the code to allow multiple robots to head towards the same beacon, thereby causing a swarm-like simulation. We will alter the code so that instead of having a unique beacon for each robot, we will generate only a few beacons and have the robots share the same exact **Beacon** object as their target destination. Here is how the data structure will change:



There will be much less **Beacon** objects created since multiple robots will share the same one. Is there any advantage to having less/shared **Beacons** ?

One important advantage to having shared beacons like this is that memory storage requirements are reduced. Imagine for example, that it takes **4** bytes each to store the **x** and **y** values of the beacon location and an additional **3** bytes to store the **color**. That means, each beacon requires a minimum of **11** bytes of storage. If we simulated **N** robots each storing their own unique beacons, this would require ((**11**+**4**) x **N**) bytes of storage in memory just to store the beacons (the extra **4** bytes storing the pointer to the beacon in the memory).

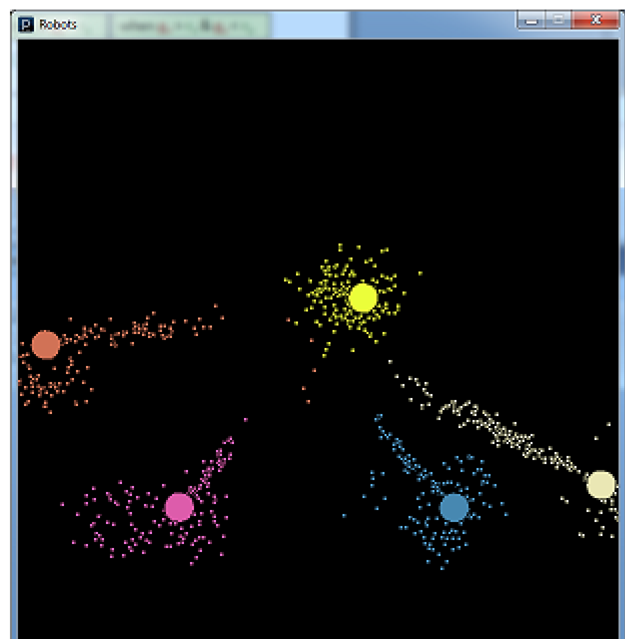
Now, assume that we create **M** beacons. This would require (**11** x **M**) bytes of storage. If each robot now kept only a single pointer (i.e., **4** bytes) to one of the **M** beacons, the total storage requirements for the beacon storage would be ((**4** x **N**) + (**11** x **M**)) bytes.

This seems a little abstract. Assume then that we have **5** beacons and **1000** robots. Then this works out to be **15,000** bytes for the non-shared beacon version and **4,055** bytes for the shared beacon version. That is a significant difference of less than **1/3** of the storage space requirements!!

This is often the reason for having shared data and shared objects ... to reduce storage space requirements for the program. However, in addition to reduced storage space, there is another important advantage to having shared beacons. If the beacon's location was to change (e.g., manually moved by the user), we would simply need to alter the (**x,y**) location of the single **Beacon** object, and since all same-swarm robots share this same **Beacon** object in memory, they will all be able to access (i.e., when heading towards) the newly changed (**x,y**) location immediately. Therefore, by changing a single variable (e.g., a beacon's **x** location), we are automatically modifying the behavior of multiple robots. This allows us to simulate the objects efficiently. Conversely, if we had stored a unique beacon object within each robot (i.e., non-shared beacons), we would need to find all robot's that share that beacon and update all of their **x** coordinates. This would be a slower process since it would require us to loop through all robots, checking their beacons for a match as to which ones the user is trying to move.

To verify this, consider altering the setup code to produce a newly-defined array of 5 beacons and then adjust the robot's to choose one of these 5 beacons as their destination. We also should adjust the **draw()** procedure to draw the beacons separately:

```
...
Beacon[]    beacons;                    // a bunch of beacons

void setup() {
  ...
  beacons = new Beacon[5];
  for (int i=0; i<beacons.length; i++)
    beacons[i] = new Beacon(random(width), random(height));

  // Make some robots with shared beacons
  robots = new Robot[1000];    // lots of robots
  for (int i=0; i<robots.length; i++) {
    robots[i] = new Robot(width/2, height/2);
    robots[i].beacon = beacons[int(random(beacons.length))];  // choose random one
  }
}

void draw() {
  background(0,0,0);

  // draw all robots
  for (int i=0; i<robots.length; i++) {
    draw(robots[i]);
    move(robots[i]);  // explained later
  }

  // draw all beacons separately
  for (int i=0; i<beacons.length; i++)
    draw(beacons[i]);
}
```



Notice how each robot is now assigned a random beacon from the **beacons** array. The program will now allow 5 groups of robots to head towards one of the 5 beacons in a swarm-like fashion.

For added enjoyment, we can add the following to allow beacons to be grabbed and moved around on the screen:

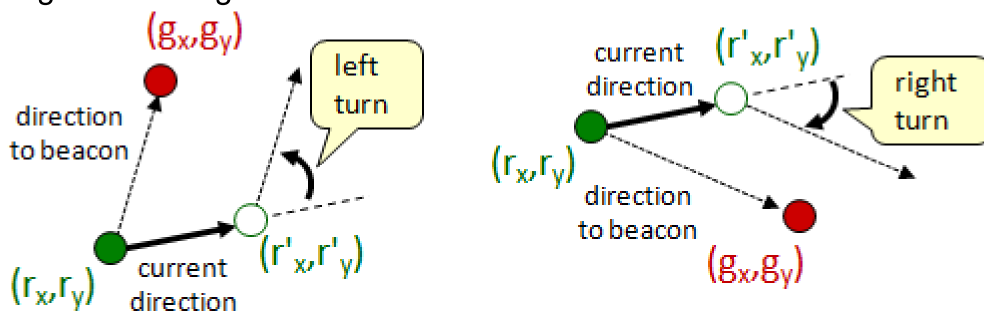
```
Beacon       grabbed;      // the beacon that has been grabbed

void mousePressed() {
  for (int i=0; i<beacons.length; i++) {
    if (dist(beacons[i].x, beacons[i].y, mouseX, mouseY) < BEACON_RADIUS) {
      grabbed = beacons[i];
      return;
    }
  }
}
void mouseDragged() {
  if (grabbed != null) {
      grabbed.x = mouseX;
      grabbed.y = mouseY;
  }
}
void mouseReleased() {
  grabbed = null;
}
```

As the beacon is moved around, you will see a subset of the robots following it around. Clearly, by allowing robot's to share **Beacon** objects we gain the advantages of saving storage space and also easy of updates when we change a beacon's location.

The only missing component of our code is the **move()** procedure...which is responsible for moving the robot towards the beacon. Recall from our ball-moving example that to move a ball (or robot) forward, we simply apply trigonometry using the robot's location and direction:

```
void move(Robot r) {
      r.x = r.x + int(r.speed*cos(r.direction));
      r.y = r.y + int(r.speed*sin(r.direction));
}
```

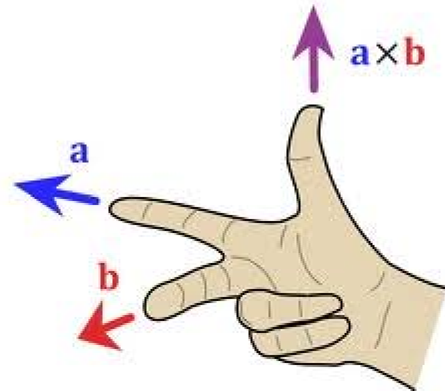
The code above simply moves the robot forward in its current direction. The harder part is to determine and update the direction of the robot so that it heads towards the beacon location. All we need to do is to look at the where the robot is heading and decide whether or not it needs to turn right or left to get towards the beacon:



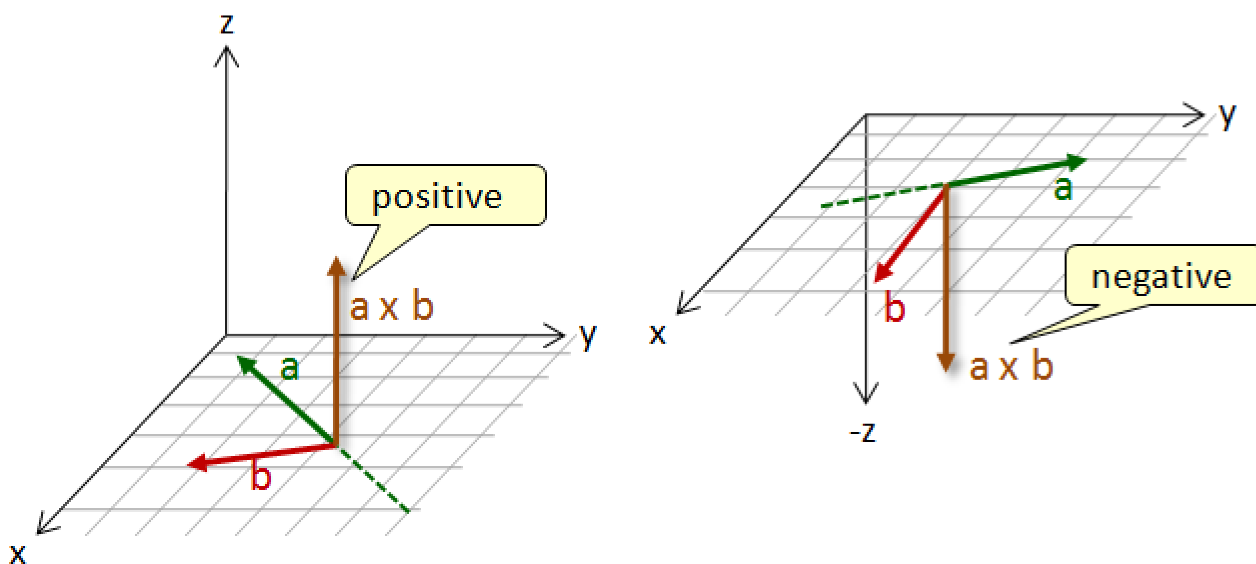
Above, ($\mathbf{r_x}$, $\mathbf{r_y}$) is the robot's current location and ($\mathbf{r'_x}$, $\mathbf{r'_y}$) would be the robot's next location if it were to travel in its current direction without turning (for this next location we could simply use the new value for **r.x** and **r.y** that we computed in our code above).

The beacon location is represented as (g_x, g_y). In order to determine whether or not the robot should take a left or right turn to get to the beacon from its current direction, we can examine the type of turn from $(r_x, r_y) \rightarrow (r'_x, r'_y) \rightarrow (g_x, g_y)$. If this is a left turn, the robot should turn left. If it is a right turn the robot should turn right. If it is a straight line, the robot will need to either move straight ahead, or straight backwards (depending on where the beacon location is).

To determine the type of turn, we can make use of the ***cross product*** of the vector from $(r_x, r_y) \rightarrow (r'_x, r'_y)$ and the vector from $(r_x, r_y) \rightarrow (g_x, g_y)$. You may recall that the cross product **a X b** is a vector that is perpendicular to the plane containing the two vectors **a** and **b**. The cross product will either be positive, negative or zero. You can visualize the cross product by using the right-hand rule as shown in the picture here.



If the cross product is positive, this indicates a left turn. If negative, then a right turn. If the cross product is zero, then there is no turn (i.e., the two vectors form an angle of 180°).



The cross product can be computed as follows:

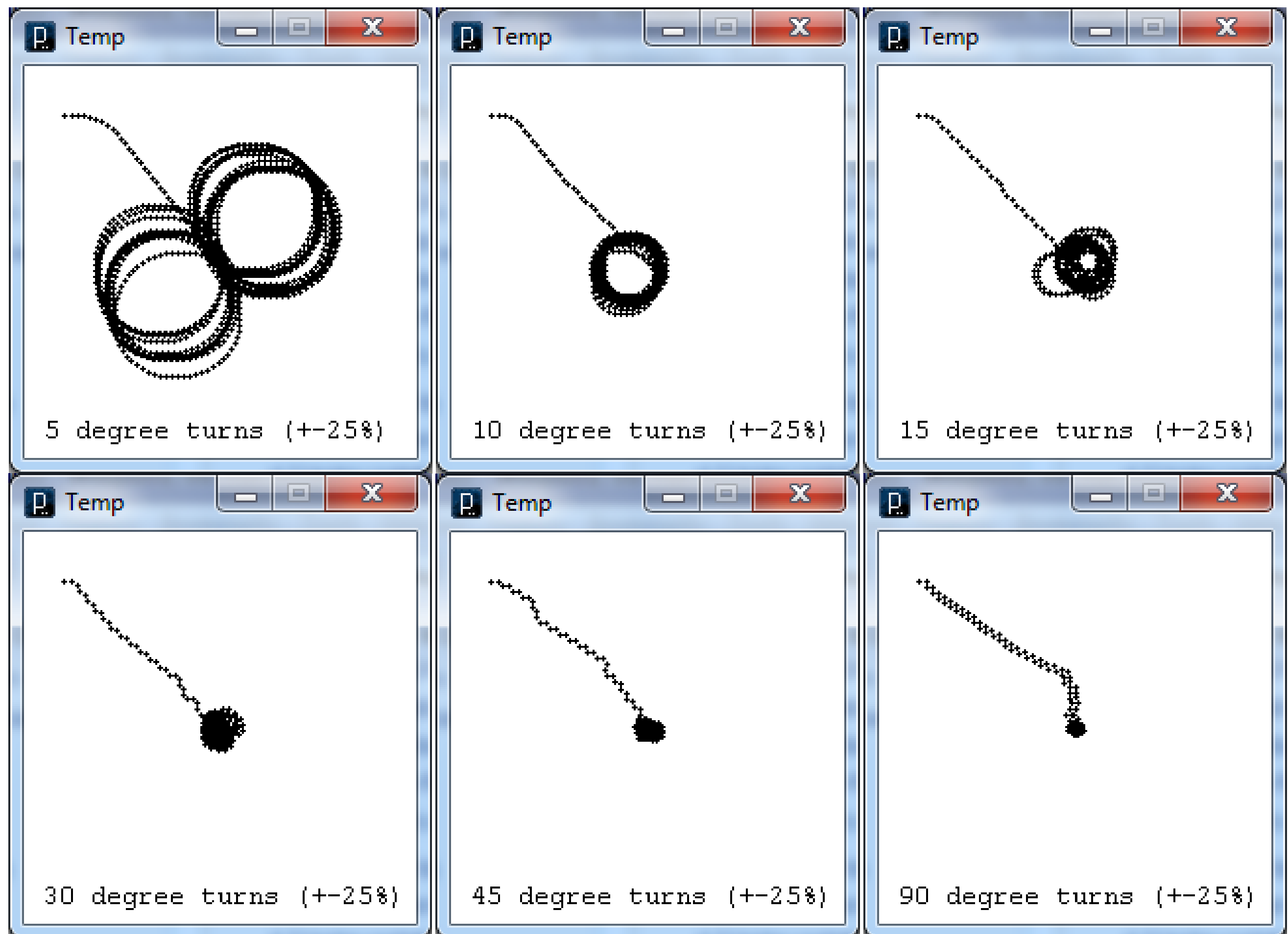
$$\text{crossProduct} = (r'_x - r_x)(g_y - r_y) - (r'_y - r_y)(g_x - r_x)$$

So, we can just plug this into our program and check for the sign of the result. Once we determine whether to turn left or right, we simply add some degree amount to the robot's direction. If we use a large degree amount (e.g., 90°) then the robot will make only right-angled turns ... not too realistic. If we use a smaller amount (e.g., 45°), then the robot will make sharp turns and will orient towards the beacon quickly. Smaller amounts (e.g., 30° or 15°) will provide a more smooth motion. Very small amounts (e.g., 1°) will cause the robot to always makes large arcing motions, taking a while to orient towards the beacon.

To make the swarm more realistic, instead of having a fixed turn amount, we could adjust it to have a ±12.5% error as follows, given a desired turn amount of **θ**:

$$amountToTurn = \theta + random(\theta/4) - (\theta/4)/2$$

As a result, here is the kind of movement pattern that a robot would produce for various values of **θ** (note that the beacon location is not displayed, but is in the center if the window):




```
void move(Robot r) {
  float nextX, nextY, crossProduct;

  nextX = r.x + int(r.speed*cos(r.direction));
  nextY = r.y + int(r.speed*sin(r.direction));

  crossProduct = (nextX - r.x)*(r.beacon.y - r.y)-(nextY - r.y)*(r.beacon.x - r.x);
  r.x = nextX;
  r.y = nextY;
  if (crossProduct < 0) // try changing to:  if((crossProduct<0)&&(random(1)<0.6))
    r.direction -= (ANGLE + random(ANGLE/4) - (ANGLE/8))/180*PI;
  else
    r.direction += (ANGLE + random(ANGLE/4) - (ANGLE/8))/180*PI;
}
```

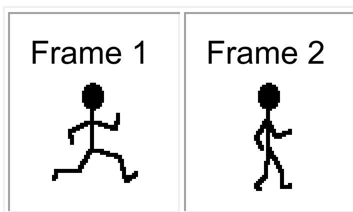
8.2 When Shared Data is Necessary

In some cases, it is absolutely necessary to share data in order to have a working program. Consider the area of computer animation. To animate something simple (i.e., 2D) in a program, programmers often use what are called ***sprites***.

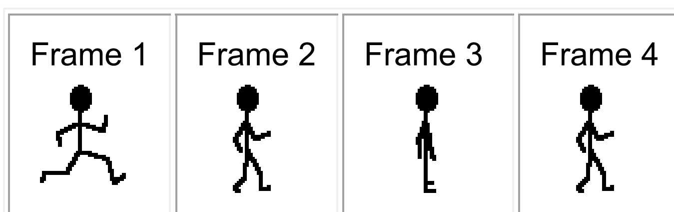
*A **sprite** is a 2D image that it integrated into a scene to form an animated character.*

Sprites are often displayed and animated by drawing a set of pictures in sequence. Each of the individual pictures, called ***frames*** represent the different "movements" of the object to be animated. So, by displaying these frames in sequence, we achieve animation.

For example, consider a stick person walking. We can do this with only two frames and just swap between them:



The animation, however, would have the undesirable characteristic of being very "jumpy", not very smooth. The problem is that there is no smooth transition between the frames. We can make a big improvement just by introducing one more picture and duplicating the 2nd frame twice to produce a 4 frame sequence:

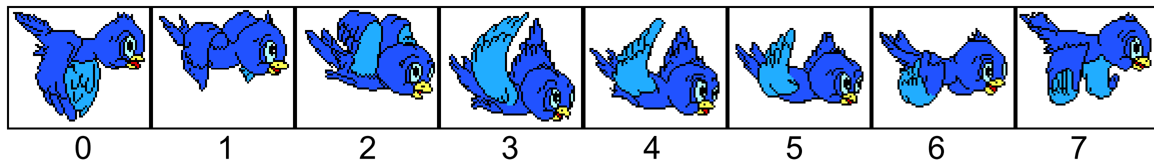


This animation would appear much smoother, but if we display the frames at the same rate, the person would appear to walk much slower. Assuming that we display 1 frame every 1/4 second, the 2-frame case would take 1/2 second to complete a cycle while in the 4-frame case, would take a full second. For the 4-frame case we can just reduce the inter-frame display delay to 1/8 of a second and the speed will then appear to match the 2-frame case. The number of frames that can be displayed in a second is known as the ***frame rate***. So, by using a delay of 1/8 seconds between frames, we have a frame rate for the animation of 8 frames per second (fps).

Example:

Consider a program that simulates birds flying across a beach scene. To make this look somewhat realistic (from a cartoon perspective), we would need to have unique pictures (i.e., frames) to display that represent the various "poses" of the bird as it flies.

We will make use of exactly 8 different poses (i.e., frames) for the birds as shown below. Each pose must be stored as a unique **.gif** image in a subfolder of our processing program's folder called **data**. The indices below the images represent the frame number:



We will need to maintain information for each bird in regards to its location on the screen, its speed and perhaps which frame is being shown at any time. Here is how we can define such a **Bird** data structure:

Define a bird object as follows:

```
class Bird {
  float      x, y;              // bird's coordinate on the screen
  float      speed;             // bird's speed in pixel movements per frame
  PImage[]   images;            // frames/pictures of the bird in various poses
  int        currentFrame;      // current frame of the bird

  Bird(float bx, float by) {
    x = bx;
    y = by;
    speed = random(10) + 5;           // random speed from 5 to 15 pixels/frame
    currentFrame = int(random(8));    // start with a random picture
    images = new PImage[8];           // array to hold 8 pictures
    for (int i=1; i<=8; i++)          // load up all 8 pictures for this bird
      images[i-1] = loadImage("Birdx" + i + ".gif");
  }
}
```

We can then create an array of **Bird** objects and display them on the screen as shown in the following code. Note that the beach image is a separate image that is displayed as a background picture. The code will display 50 birds at random locations on the window at some random pose. Even though the birds are displayed, they are not moving...they seem still. The **frameRate** is the number of times per second that the **draw()** procedure is called. It is set to 20 and this will be fast enough to animate the birds, as we will see later. If it is not set, then the **frameRate** will be too fast because the default rate is 60 fps.

```
Bird[]        birds;
PImage        beach;

void setup() {
  size(800,600);
  birds = new Bird[50];
  for (int i=0; i<8; i++)
    birds[i] = new Bird(random(width),random(height/2));
  beach = loadImage("beach.jpg");
  frameRate(20);
}

void draw() {      // This is called repeatedly at a fixed frameRate
  image(beach, 0, 0);
  for (int i=0; i<8; i++)
    draw(birds[i]);
}

void draw(Bird b) {
  image(b.images[b.currentFrame], b.x, b.y);
}
```

To animate the birds, we must now do two things: (1) Have them vary their images, and (2) have them move horizontally. We can update the **draw()** procedure by adding a **move()** procedure as follows:

```
void draw() {      // This is called repeatedly at a fixed frameRate
  image(beach, 0, 0);
  for (int i=0; i<8; i++) {
    move(birds[i]);
    draw(birds[i]);
  }
}
```

The **move()** procedure must move the bird forward and this can be done simply by adding the bird's speed to the x coordinate, making sure that when it reaches the right side of the window, we cause a wraparound to the left side again. To do this, we simply set the **x** value to an amount which is **-width** of the bird image so that the bird slowly flies back onto the window again from the left. To change the bird's image, we simply increment the **currentFrame**, making sure that when it reaches 8, we set it back to 0 again ...using the modulus operator:

```
void move(Bird b) {
  // Move the bird 10 pixels forward
  b.x = b.x + b.speed;

  if (b.x > width)
    b.x = -102; // 102 is the width of the Bird image

  b.currentFrame = (b.currentFrame + 1) % 8;
}
```

Now the birds will appear to be flying nicely. Try changing the array to make **500** birds. Depending on your computer's memory ... the code may or may not run. My computer produced an error indicating that I would need to increase the memory setting in the **Preferences** under the **File** menu.

The problem is simple. The current code loads 8 images for each bird. Therefore, 500 birds would require (500 x 8 = 4000) images to be loaded and stored in the program. However, it is easily seen that the birds are all using the same 8 image files. Therefore, instead of having each bird store their own images, it makes more sense to simply load the 8 images and have the birds "share" the images. We can add this to the program:

```
PImage[]      birdFrames;
```

and then add this to the **setup()** procedure:

```
birdFrames = new PImage[8];
for (int i=1; i<=8; i++)
      birdFrames[i-1] = loadImage("Birdx" + i + ".gif");
```

Then we can remove the **images** from the **Bird** object:

```
class Bird {
  float      x, y;
  float      speed;
  int        currentFrame;

  Bird(float bx, float by) {
    x = bx;
    y = by;
    speed = random(10) + 5;
    currentFrame = int(random(8));
  }
}
```

Finally, we make changes to the **draw(Bird b)** procedure:

```
void draw(Bird b) {
  image(birdFrames[b.currentFrame], b.x, b.y);
}
```

Now, you may even be able to add up to 50,000 birds without problems...although the program would become very slow ;)

So in this example, we see that sharing is sometimes necessary in order to reduce memory space requirements and have a running program. There are also examples in which sharing is NOT even desired at all...

8.3 Separating Shared Data Again

Example:

Even though sharing objects may be to our advantage, there are some situations where we do not want shared objects. For example, sometimes we need to make copies of objects and we would like the copy to be a truly unique copy so that we can leave the original intact and safe at all times. We may think of making photocopies of important documents so that the original document can be stored away safely.

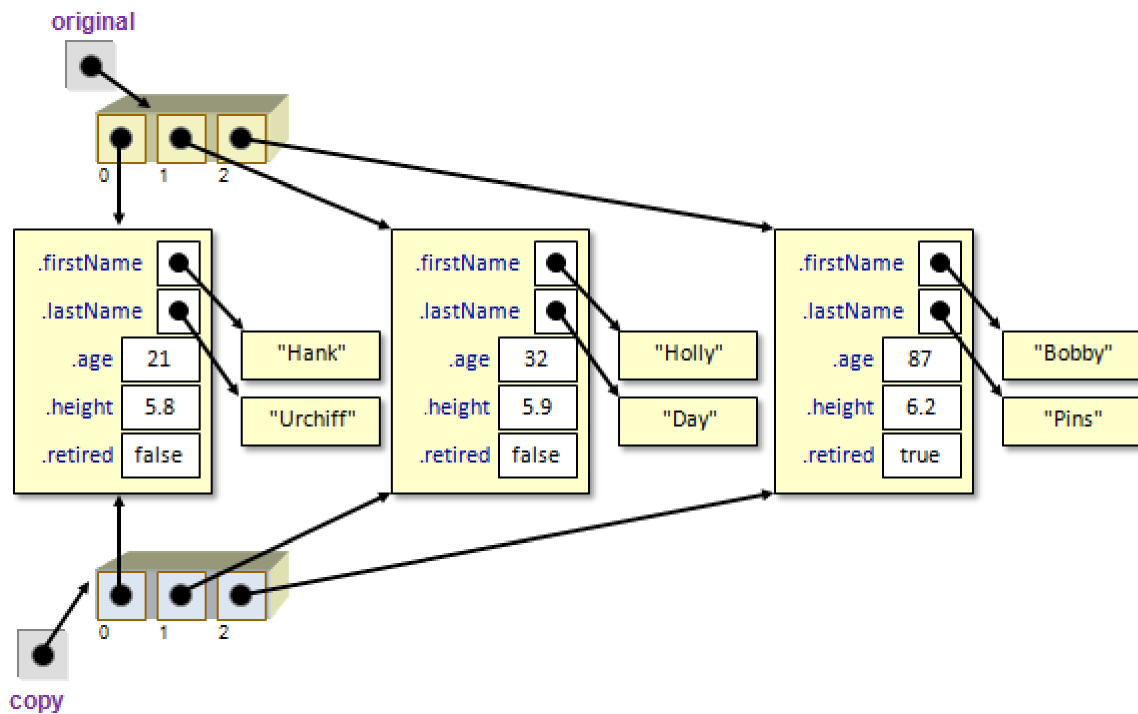
Consider the following function which creates and returns a copy of a list:

```
Function:  makeCopy(aList)

1.    anotherList ← new array with capacity aList.length
2.    for index i from 0 to aList.length - 1
3.        anotherList[i] ← aList[i]
4.    return anotherList
```

The code produces a new array containing the objects from the original list. Consider testing this function with an array of **Person** objects, which have a firstName, lastName, age, height and retiredStatus as follows:

```
Algorithm:  CopyTest1

1.    original ← new array with capacity 3
2.    original[0] ← new Person("Hank", "Urchif", 21, 5.8, false)
3.    original[1] ← new Person("Holly", "Day", 32, 5.9, false)
4.    original[2] ← new Person("Bobby", "Pins", 87, 6.2, true)
5.    copy ← makeCopy(original)
```

Here is what we accomplish with this code …



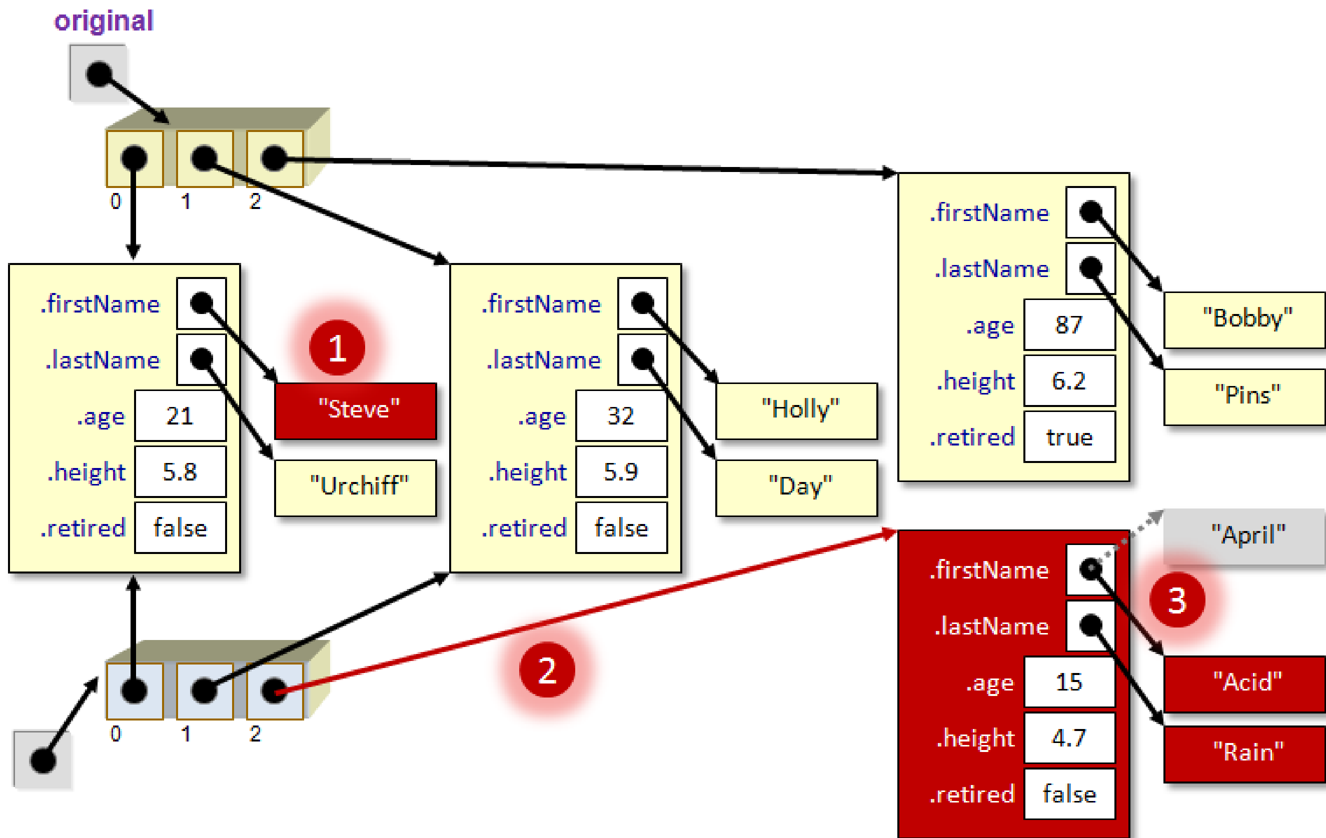
Notice that when we add the **Person** objects to the copy, they are actually shared between the two arrays. This is known as a ***shallow copy***. The advantage is that we do not need to use up additional memory space to store duplicate information for the copy. That is, we do not need to store the first and last names twice (i.e., for the original and the copy) since they are the same names. Normally this is not a problem when writing code as long as we know that the items are shared.

We need to understand the consequences of having such shared data. Consider what happens in the following example as we make changes to the copy:

Algorithm: CopyTest2

```
1.   original ← new array with capacity 3
2.   original[0] ← new Person("Hank", "Urchif", 21, 5.8, false)
3.   original[1] ← new Person("Holly", "Day", 32, 5.9, false)
4.   original[2] ← new Person("Bobby", "Pins", 87, 6.2, true)

5.   copy ← makeCopy(original)

6.   copy[0] .firstName ← "Steve"
7.   copy[2] ← new Person("April", "Rain", 15, 4.7, false)
8.   copy[2] .firstName ← "Acid"
```

We are doing three things to the copy: (1) changing a person's name, (2) replacing one person entirely with a new one, and (3) changing the new one's name. Here is what happened:



When changing Hank's name in the copy to "Steve", the original list is modified as well since the **Person** object, that both the original and copy lists were sharing, has now been altered. This can have serious consequences in your program since you may end up with a part of your code that unknowingly affects other parts of your program by modifying data structures that were not meant to be changed.

Some modifications, however, can be made to the copy that do not affect the original. For example, when replacing Bobby with the new person **April** in the **copy**, the original remains unchanged since we are simply moving a "pointer" in the copy to a new memory location. Then, when we replace April's name with "Acid", the original is not affected since nowhere does it refer to that newly create **Person** object.

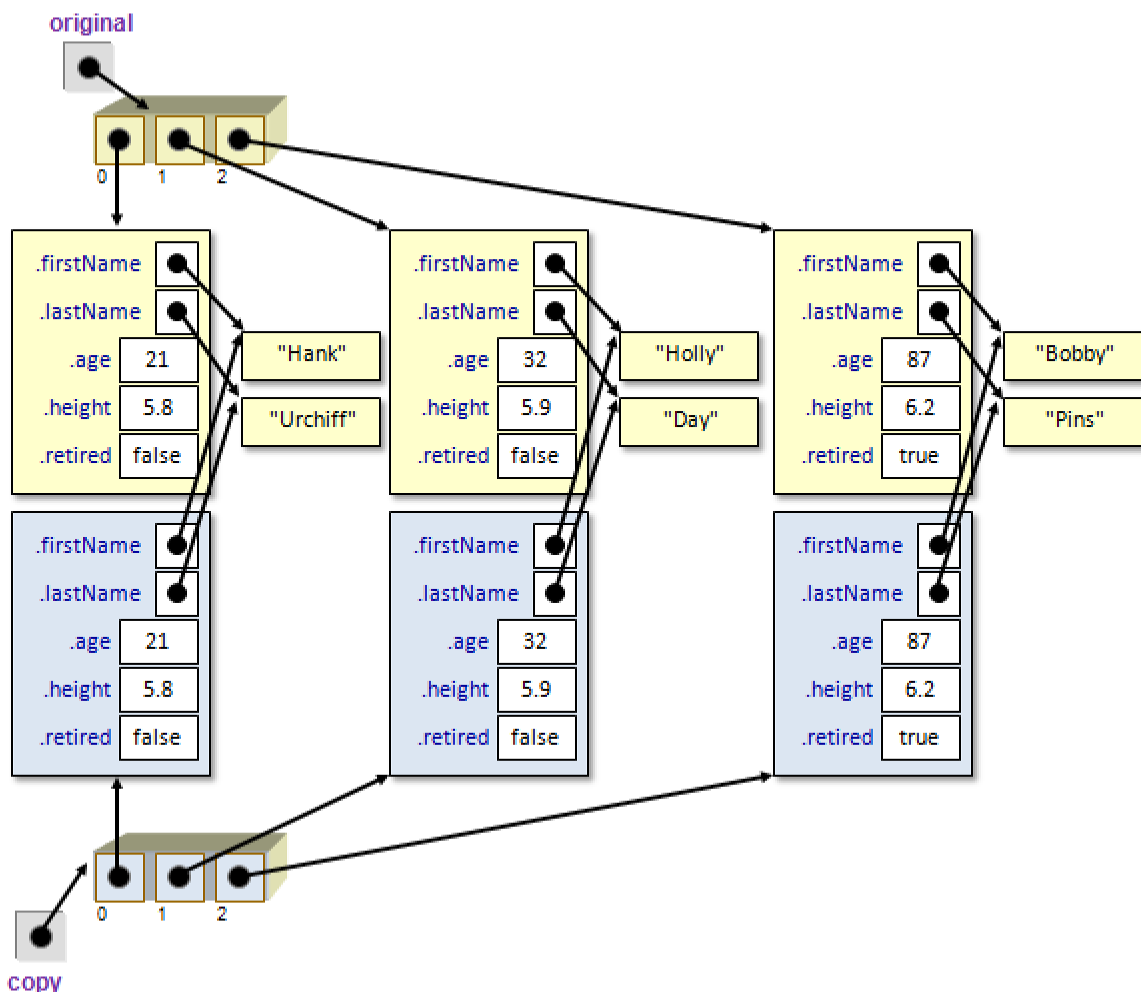
So you can see, by replacing, adding to or removing from a copied list, the original list remains intact. However, when we access a shared object from the copy and go into it to make changes to its attributes, then the original list will be affected.

To make a more thoroughly-separated copy, we could make what is known as a ***deep copy*** of the list so that the **Person** objects are not shared between them (i.e., each list has their own copies of the **Person** objects). To do this, we would need to make copies of the objects in the lists. That means, we would need to create a new object for each item in the list and then copy over all of the attributes so that they match the original objects.

Here is how we can modify the **makeCopy** function to accomplish this:

```
Function:  makeCopy(aList)

1.      anotherList ← new array with capacity aList.length
2.      for index i from 0 to aList.length - 1 {
3.              anotherList[i] ← new Person
4.              anotherList[i].firstName ← aList[i].firstName
5.              anotherList[i].lastName ← aList[i].lastName
6.              anotherList[i].age ← aList[i].age
7.              anotherList[i].height ← aList[i].height
8.              anotherList[i].retiredStatus ← aList[i].retiredStatus
        }
9.      return anotherList
```

Notice that much more work is involved. However, the effect of running our **CopyTest1** algorithm would be as follows:

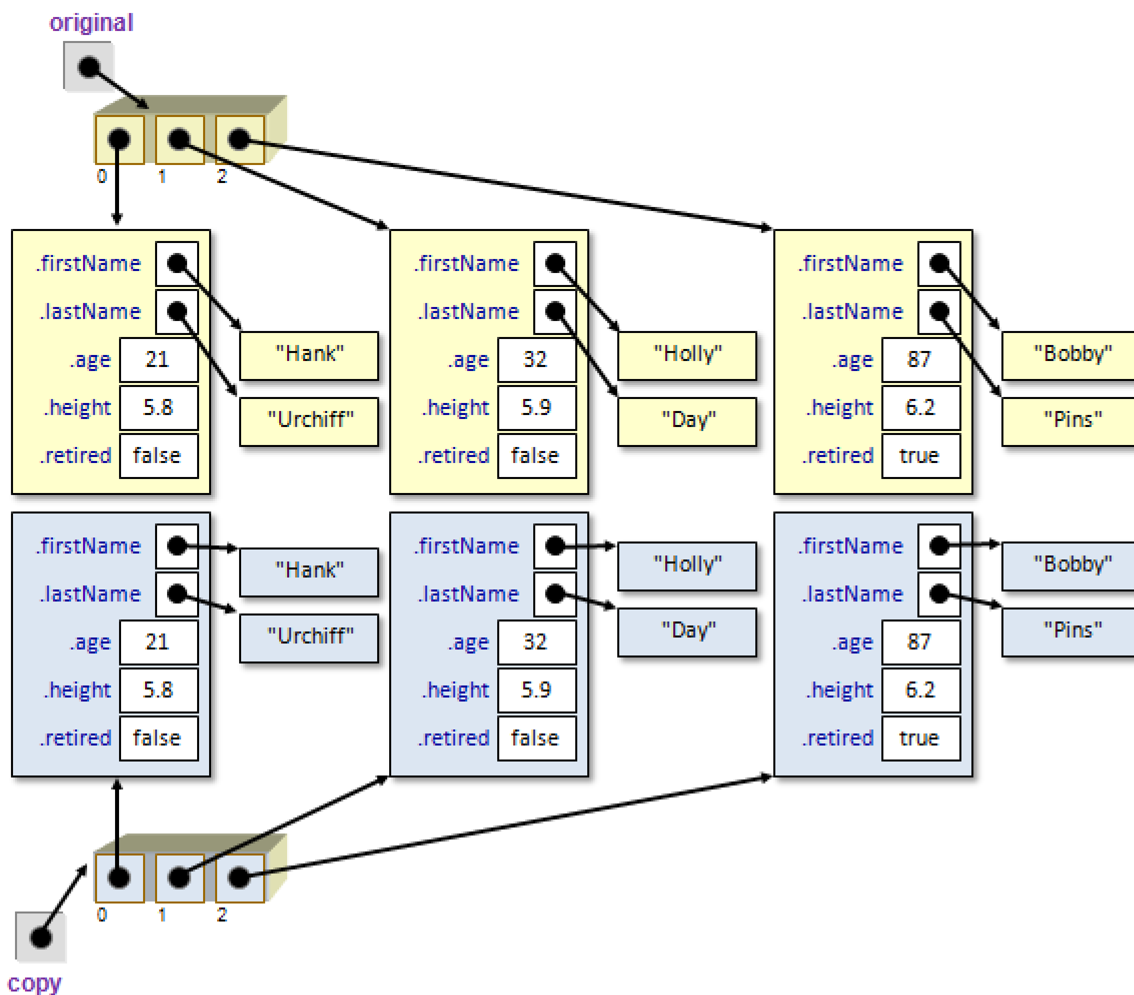


Notice that there are unique **Person** objects now and that changing the name of anyone in the copy will not affect the original. You may notice though, that the Strings are still shared! That means, if we altered any characters in one person's name in the copy, this would affect the original (i.e., remember...***replacing*** shared objects does not cause problems but ***modifying*** shared objects does cause a problem).

Fortunately, in Processing (and JAVA), it is not possible to modify characters in a String, so this would never be a problem. However, to be safe, a truly deep copy can be made by copying these strings as well by changing lines 4 and 5 in the **makeCopy** function to:

```
4.    anotherList[i].firstName ← new String with same characters as aList[i].firstName
5.    anotherList[i].lastName ← new String with same characters as aList[i].lastName
```

Then, we would have a true copy:



In fact, in order to make a truly deep copy, we would need to ensure that we ***thoroughly*** copy each of the attributes of all objects. That is, if an object is made up of other objects ... we must go into those other objects and make deep copies of them as well.

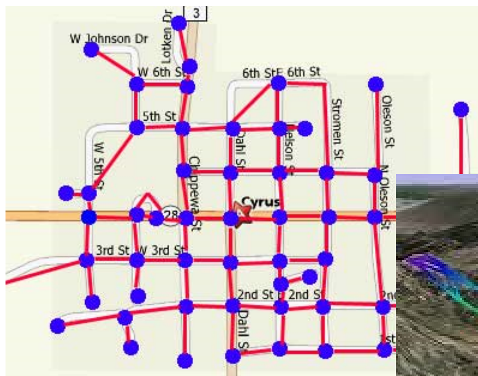
8.4 Example: Graph Editor

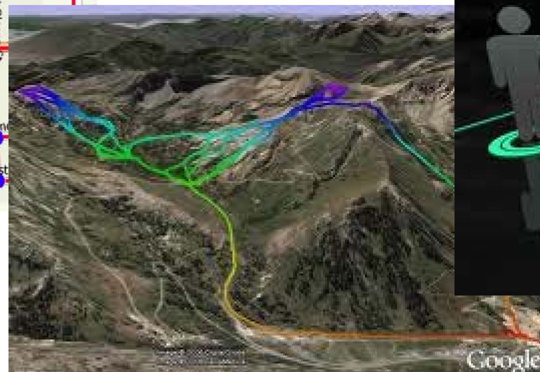
As a final example, we will consider an application in which data is shared within a data structure called a ***graph***:

> *A **graph** is an abstract data structure that represents interconnectivity of a set of objects (called **nodes** or Vertices). Two nodes of a graph are connected by an **edge** to indicate some kind of relationship between the nodes.*

Graphs are used throughout computer science. They can be used to store data that represents interconnectivity between various "things". For example, graphs can be used to represent

- connected flights from one city to another
- roads and intersections on a map
- various pathways on a terrain (e.g., water flow)
- interconnectivity in networks (e.g., social networks)
- etc...

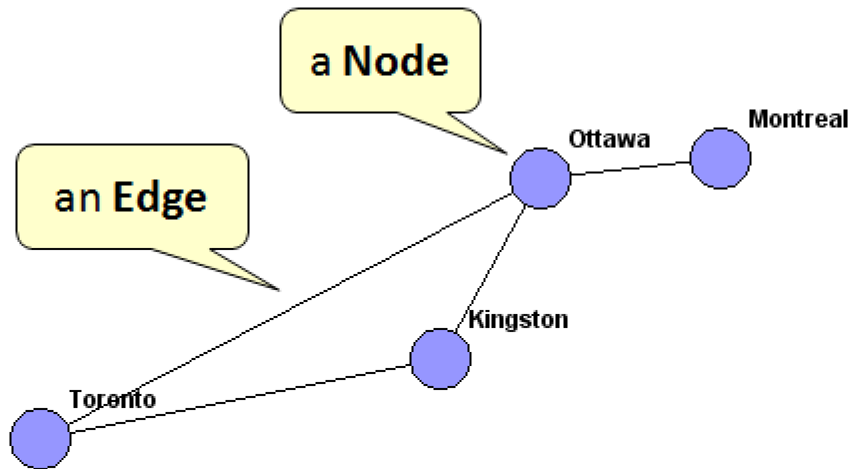




Regardless of the application, graphs can be automatically generated or created manually. We will examine a simple application that allows the user to create, manipulate and edit a graph on a window in Processing.

This larger programming example with bring together many of the concepts that you have learned in the course including the use of data structures, graphics, event handling, arrays, searching and the sharing of data.

To begin, here is what the graph will look like:



We will then be able to move nodes and edges around the graph as well as add and delete nodes and edges. As you will see, the edges of the graph will share nodes so that when edges are moved, their nodes will also move and when a node is deleted, all edges connected to that node will be deleted.

We need to begin by understanding how the graph is stored. The graph will be made up of **Node** objects which have a particular (**x**,**y**) location on the screen as well as a **label**:

```
class Node {
  int         x, y;
  String      label;

  Node(int ix, int iy, String lab) {
    x = ix;
    y = iy;
    label = lab;
  }
}
```

Creating nodes is therefore quite easy, since we simply call the constructor as follows:

```
new Node(350, 250, "Ottawa");
```

Edges will connect two nodes. It does not make sense for an edge to exist unless it connects two nodes of the graph. For example, all of an airline's flights have a departure city and an destination city. It does not make sense for either one of these cities to be missing. Therefore, an edge will not connect two *points* on the screen, rather it will connect two nodes of the graph. Here is the definition of the **Edge** data structure:

```
class Edge {
  Node        startNode, endNode;

  Edge(Node start, Node end) {
    startNode = start;
    endNode = end;
  }
}
```

Notice how the attributes of the **Edge** are actually **Node** objects. Creating an edge therefore requires us to already have two **Node** objects:

```
Node n1 = new Node(350, 250, "Ottawa");
Node n2 = new Node(100, 380, "Toronto");

Edge e = new Edge(n1, n2);
```

Now, for the **Graph** itself ... it is really nothing other than a set of nodes and edges. We can store these using two arrays ... one for the nodes and one for the edges. But how big should the arrays be ? Ideally, we should allow them to grow (recall from chapter 5 how we were able to grow an array by making a new one and copying over the contents). However, to keep our code simple, we will not allow the arrays to grow but instead we will set a maximum number of nodes and edges that can be added to the graph. Here is our definition for the **Graph** data structure:

```
final int MAX_GRAPH_NODES = 100;      // arbitrarily chosen #
final int MAX_GRAPH_EDGES = 1000;     // arbitrarily chosen #

class Graph {
  int       numNodes, numEdges;
  Node[]    nodes;
  Edge[]    edges;

  Graph() {
    nodes = new Node[MAX_GRAPH_NODES];
    edges = new Edge[MAX_GRAPH_EDGES];
    numNodes = numEdges = 0;
  }
}
```

Graphs with no nodes or edges are made by simply calling the constructor: `new Graph();`

However, in order to make an interesting graph, we will need the ability to add nodes and edges to it. We can write simple procedures that allow us to add nodes and edges to the corresponding graph arrays as follows:

```
// Add the given node to the given graph
void addNode(Graph g, Node n) {
  if (g.numNodes < MAX_GRAPH_NODES) {
    g.nodes[g.numNodes] = n;
    g.numNodes++;
  }
}

// Add an edge to the given graph from node a to node b
void addEdge(Graph g, Node a, Node b) {
  if (g.numEdges < MAX_GRAPH_EDGES) {
    g.edges[g.numEdges] = new Edge(a, b);
    g.numEdges++;
  }
}
```

Notice the error checking to make sure that we do not go past the maximum array sizes.

Here is a simple function that creates and returns the graph shown earlier:

```
Graph exampleGraph() {
  Graph g = new Graph();
  Node n1 = new Node(350, 250, "Ottawa");
  Node n2 = new Node(100, 380, "Toronto");
  Node n3 = new Node(300, 340, "Kingston");
  Node n4 = new Node(440, 240, "Montreal");
  addNode(g, n1);
  addNode(g, n2);
  addNode(g, n3);
  addNode(g, n4);
  addEdge(g, n1, n2);
  addEdge(g, n1, n3);
  addEdge(g, n1, n4);
  addEdge(g, n2, n3);

  return g;
}
```

To display the graph, we need to write the **setup()** procedure that will define the window size, initialize the graph and prepare the font for the node labels:

```
Graph    graph;       // Global variable to store the graph

// Start the program by initializing everything
void setup() {
  size(600,600);
  graph = exampleGraph();

  // Set up the font for the node labels (must be in data directory)
  PFont font = loadFont("Arial-BoldMT-12.vlw");
  textFont(font);
}
```

The above code makes a 600x600 window and loads a font that will be used to draw the labels onto the nodes. It also creates the example graph and stores it in a global variable that we can access from anywhere within our program. (Note that the font string shown here was automatically produced from the **Tools/Create Font...** menu option in Processing).

Drawing a node **n** is as simple as calling the **ellipse()** procedure where the node's (x,y) position is its center and the NODE_DIAMETER is the width and height of the node:

```
final int NODE_RADIUS = 15;
final int NODE_DIAMETER = NODE_RADIUS*2;
ellipse(n.x, n.y, NODE_DIAMETER, NODE_DIAMETER);
```

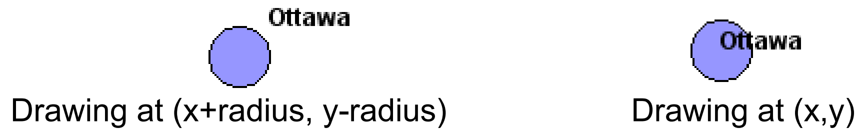
Drawing an edge **e** is also simple using the **line()** procedure:

```
line(e.startNode.x, e.startNode.y, e.endNode.x, e.endNode.y);
```

To draw the label of node **n**, we simply use the **text()** procedure:

```
text(n.label, n.x + NODE_RADIUS, n.y - NODE_RADIUS);
```

Notice how we added the node's **radius** to the **x** and subtracted from the **y**. This allows us to draw the label up and to the right of the node ... making it more readable:



Drawing at (x+radius, y-radius) Drawing at (x,y)

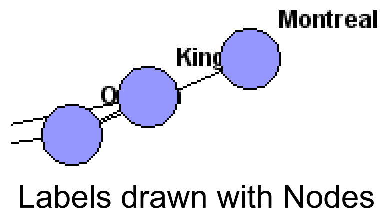
Here is the **draw()** procedure that draws the whole graph altogether:

```
void draw() {
  background(255);    // Paint the background white
  stroke(0,0,0);      // black border

  // Draw the Edges
  for (int i=0; i<graph.numEdges; i++) {
    Edge e = graph.edges[i];
    line(e.startNode.x, e.startNode.y, e.endNode.x, e.endNode.y);
  }

  // Draw the Nodes
  for (int i=0; i<graph.numNodes; i++) {
    Node n = graph.nodes[i];
    fill(150,150,255);         // light blue inside color
    ellipse(n.x, n.y, NODE_DIAMETER, NODE_DIAMETER);
  }

  // Draw the labels on the nodes
  fill(0);
  for (int i=0; i<graph.numNodes; i++) {
    text(graph.nodes[i].label,
         graph.nodes[i].x + NODE_RADIUS,
         graph.nodes[i].y - NODE_RADIUS);
  }
}
```

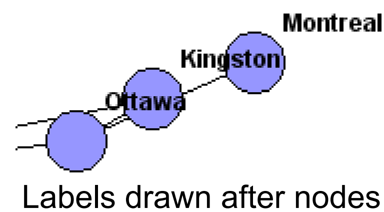
Notice that the code first clears the background, then draws the edges, then the nodes and finally the labels of the nodes. Drawing the edges first prevents the edges from drawing on top of the center of the nodes:

Edges drawn first Nodes drawn first

Likewise, the labels of the nodes are drawn afterwards so that they will always appear on top and not get hidden underneath the other nodes:



Labels drawn with Nodes



Labels drawn after nodes

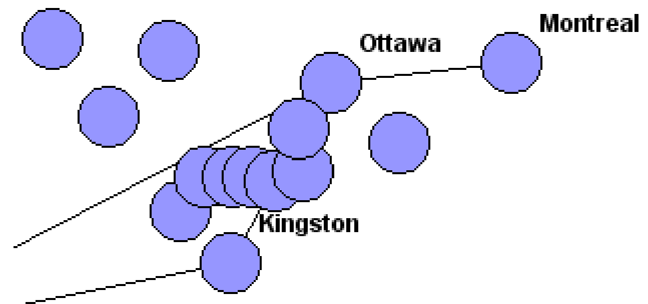
Now that we have the ability to draw the graph, we can start working on allowing it to be edited and manipulated.

Adding Nodes:

To begin, we will write code that allows the user to add a node to the graph by double-clicking the left mouse button at some location in the window. The new node should appear centered where the user clicked. We need to write a **mouseClicked()** event handler:

```
void mouseClicked() {
  addNode(graph, new Node(mouseX, mouseY, ""));
}
```

Notice how simple the code is. We simply check if the mouse was clicked and add a node to the graph with the current mouse position as the node's center. We can now add new nodes to the graph ... as shown here →



To adjust the code so that the node only gets added when the mouse is double-clicked, we need to make use of a variable in Processing called **mouseEvent**. That variable is an object that allows us to apply various functions to it. In the next course, we will see that we can "attach" (or associate) functions and procedures to objects. In this case, we will call the **getClickCount()** function which returns the number of successive clicks that the user made so that we can add the node only when a 2nd successive click was made:

```
void mouseClicked() {
  if (mouseEvent.getClickCount() == 2)
    addNode(graph, new Node(mouseX, mouseY, ""));
}
```

Selecting Nodes:

Many drawing programs have the ability to select and move objects around. We will now add functionality to allow our nodes to be selected. Selected nodes should *appear* selected somehow ... in our case ... we will simply color them red instead of blue.

Rather than trying to keep a list of selected nodes, we can alter the **Node** data structure to include a **boolean** so that each node has the ability to be selected or unselected:

```
class Node {
  int        x, y;
  String     label;
  boolean    selected;

  Node(int ix, int iy, String lab) {
    x = ix;
    y = iy;
    label = lab;
    selected = false;
  }
}
```

So then what do we do to select a node ? Likely we will double-click on the node. What if we double-click on a node that is already selected ? It should become unselected. So, double-clicking on a node should cause it to toggle between being selected and unselected.

We need to modify our **mouseClicked()** event handler. There is a slight problem ... double-clicking anywhere on the window will cause a new node to be added. We need to adjust the code therefore so that when we double-click on top of a node ... we select it instead of adding a new node.

```
if (there is a node already at the mouse location)
     select the node
otherwise
     add a new node at the mouse location
```

So, we need to find out whether or not there is a node at the mouse location. We can write a function to find and return the node at a given location. To do this, we need to look at all the nodes and determine which one was clicked on (if any). Here is one way to do this:

```
Node nodeAt(Graph g, int x, int y) {
  for (int i=0; i<g.numNodes; i++) {
    if ((g.nodes[i].x == x) && (g.nodes[i].y == y))
      return g.nodes[i];
  }
  return null;
}
```

This code will look through all the nodes in the **graph** and find the one whose center matches the point specified by the given (**x,y**) point. If we find such a node, it is returned right away, otherwise the FOR loop completes ... no matching node was found ... so the function returns **null**.

The problem with this code, however, is that we are extremely unlikely to click on the exact center of the node. It would be very difficult and painfully tedious to force the user to click on the center of a node to select it. So, we need to adjust our code so that the user may click anywhere inside the node to select it. How do we determine whether the point (**x,y**) is inside the node or outside of it ? We already solved this problem when we simulated the throwing of a ball back in chapter 3. We just need to check the distance from the click point to the center of the node and ensure that it is less than the radius of the node:

```
Node nodeAt(Graph g, int x, int y) {
  for (int i=0; i<g.numNodes; i++) {
    Node n = g.nodes[i];
    float d = sqrt(((n.x - x) * (n.x - x)) + ((n.y - y) * (n.y - y)));
    if (d <= NODE_RADIUS)
      return n;
  }
  return null;
}
```

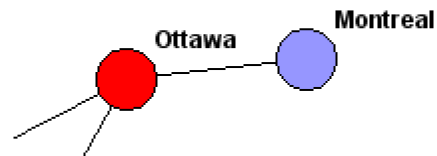
Note that the code above can be made more efficient by checking the "square" of the distance against the square of the radius to avoid the computation of a square root ... thereby speeding up the computation.

Now we can adjust the **mouseClicked()** event handler to allow nodes to be selected:

```
void mouseClicked() {
  if (mouseEvent.getClickCount() == 2) {
    Node n = nodeAt(graph, mouseX, mouseY);
    if (n != null)
      n.selected = !n.selected;
    else
      addNode(graph, new Node(mouseX, mouseY, ""));
  }
}
```

All that remains to be done is to draw nodes as selected. We can replace the `fill(150, 150, 255)` line in the **draw()** procedure by the following:

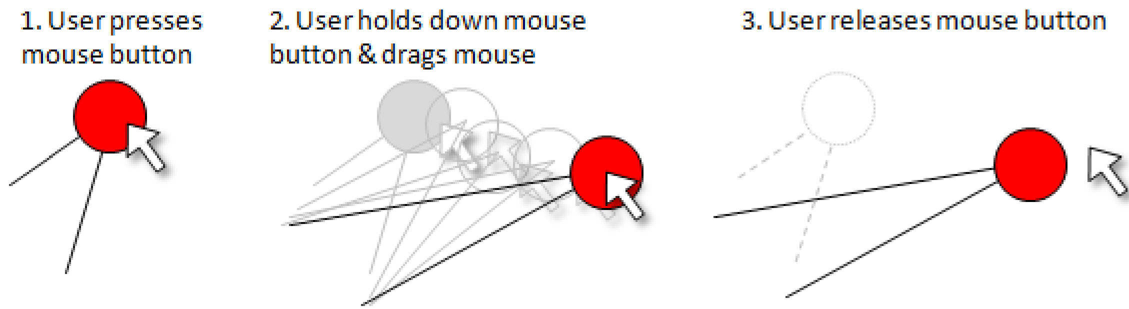
```
if (n.selected)
  fill(255,0,0);     // red inside
else
  fill(150,150,255); // light blue inside
```



We can now select nodes.

Moving Nodes:

A natural user action to move nodes around on the window would be to select a node and then grab it and drag it to its new position. To allow this functionality, we will need to incorporate into our program additional event handlers. Let us consider what happens when the user drags a node:



So, we will need to write **mousePressed**, **mouseDragged** and **mouseReleased** event handlers. What happens when the mouse is pressed on a selected node ? Well, the node does not move. Nothing really "happens" visually. What about when the mouse is dragged ? Well, we need to move the node that we had clicked on to the new mouse location. Within the **mouseDragged** event handler, we will need to know which node we had initially pressed the mouse on. So, in the **mousePressed()** event handler all we need to do is to find out which node was at that position and remember it. We can store this into a variable called **dragNode** as follows:

```
Node      dragNode;     // Add to the top of program

void mousePressed() {
  dragNode = nodeAt(graph, mouseX, mouseY);
}
```

Then, as the mouse is dragged, we simply access this **dragNode** and set its (x,y) location to be the mouse's location as follows:

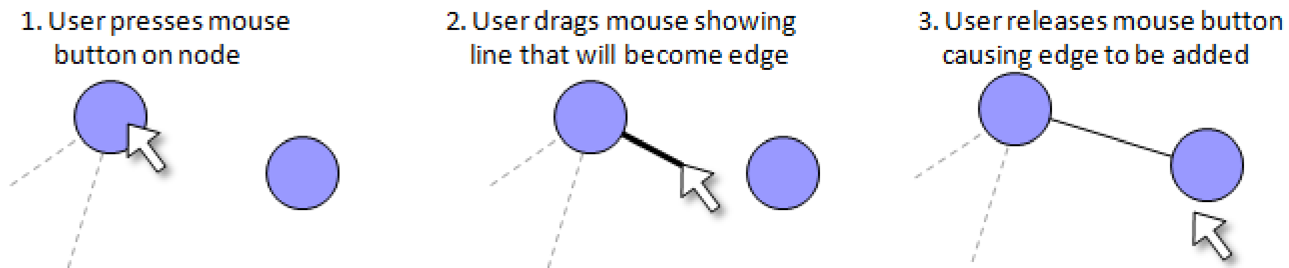
```
void mouseDragged() {
  dragNode.x = mouseX;
  dragNode.y = mouseY;
}
```

However, it is possible that the user may drag the mouse without first having pressed on a node. In that case, the **nodeAt()** function will return **null** (as it is supposed to) and our code would stop working since **dragNode** would be **null** and we cannot access **null.x** or **null.y**. So we should check to ensure that **dragNode** is not **null**. Also, we should ensure that the **dragNode** is **selected**, otherwise we may be moving nodes that we did not intend to move. Here are the changes:

```
void mouseDragged() {
  if (dragNode != null) {
    if (dragNode.selected) {
      dragNode.x = mouseX;
      dragNode.y = mouseY;
    }
  }
}
```

Adding Edges:

What kind of user-action will our program consider for adding edges ? There are a few options. For example, we could write code that allows an edge to be added simply by clicking two nodes one after another. However, a "nicer" interface will allow the user to click on a node and "drag" a line towards another edge so that when the user lets go on another node, an edge is added:



The idea is similar to that of "stretching" an elastic band tied from the center of the start node with the other end on the mouse cursor. This ***elastic-banding*** effect is common with many drawing programs as it provides feedback to the user giving a rough idea as to how the edge is going to appear in the graph.

To accomplish this, as the mouse is being dragged, we will need to draw a line from the start node to the mouse's current location. Where do we do the drawing of this line ? It needs to be done in the **draw()** procedure. So, in the **draw()** procedure we will need to know the start node as well as the current mouse location.

The code in the **mousePressed** event handler already stored the start node as **dragNode**. All we would then need to do is to ensure that as the user drags the mouse (i.e., in the **mouseDragged** event handler) we store the mouse location. We could use variables called **elasticEndX** and **elasticEndY** as follows:

```
int     elasticEndX, elasticEndY;     // Add to the top of program

void mouseDragged() {
  if (dragNode != null) {
    if (dragNode.selected) {
      dragNode.x = mouseX;
      dragNode.y = mouseY;
    }
    else { // store end point for edge to be added
      elasticEndX = mouseX;
      elasticEndY = mouseY;
    }
  }
}
```

Then, in the **draw()** procedure we add code to ensure that the line is drawn. However, we want to make sure that the user first clicked on a node to begin the edge adding process.

If the user clicked off of a node, then we can detect this by checking if **dragNode** is **null** as follows:

```
// Draw the elastic band
stroke(0,0,0); //black
if (dragNode != null)
 line(dragNode.x, dragNode.y, elasticEndX, elasticEndY);
```

This code should appear after **background(255)** but before the nodes of the graph are drawn so that the line will appear behind the nodes and not cut across them.

If the user decides to start drawing an edge but then lets go of the mouse button somewhere other than on another node, then the edge should not be added and the line should disappear. However, our code does not do this. We need to have a way of informing the **draw()** procedure NOT to draw the line once the user releases the mouse button. The simplest way to do this is to set the **elasticEndX** to have a value of **-1**... or something similar ... in the **mouseReleased** event handler. Then we can check for this in the **draw()** procedure:

```
void mouseReleased() {
  elasticEndX = -1;
  elasticEndY = -1;
}

void draw() {
  ...
  if ((dragNode != null) && (elasticEndX != -1))
    line(dragNode.x, dragNode.y, elasticEndX, elasticEndY);
  ...
}
```

Of course, we must handle the case where the user successfully releases the mouse on a node. Then we need to add a new edge. We should add the edge as long as the user released the mouse on a "different" node...not the same one as the dragNode:

```
void mouseReleased() {
  elasticEndX = -1;
  elasticEndY = -1;
  if (dragNode != null) {
    Node n = nodeAt(graph, mouseX, mouseY);
    if ((n != null) && (n != dragNode))
      addEdge(graph, dragNode, n);
    }
  }
}
```

We now have the ability to add nodes to the graph. You may notice that dragging a "selected" node does not add an edge but still allows the user to move the node around.

Deleting Nodes:

Of course, sooner or later we are going to create some nodes by mistake and want to delete them. With many drawing programs, a common action for deleting objects is to first select them and then press delete.

How do we allow them to be deleted ? We allow the user to press the **DELETE** key. This should cause them to be deleted. We simply write a **keyPressed()** event handler and determine if the **key** variable has been set to **DELETE**. Then we delete any nodes that have been selected:

```
void keyPressed() {
  if (key == DELETE) {
    for (int i=0; i<graph.numNodes; i++) {
      if (graph.nodes[i].selected) {
        deleteNode(graph, graph.nodes[i]);
        i--; // required so that we check the same index next time
      }
    }
  }
}
```

Notice that that the code loops through all nodes, determines if they are selected and then calls a **deleteNode()** procedure (we need to write this yet). The **i--** will make sense after we write the **deleteNode()** procedure.

Deleting a node is actually a complicated process since there may be edges connected to a node. We cannot allow any edges to remain in the graph if the edge's **startNode** or **endNode** has been deleted. So we first need to delete all the edges connected to the node being deleted, then we can delete the node itself. We can write a procedure that loops through the edges and deletes the ones that have the given node as its start or end node as follows:

```
void deleteNode(Graph g, Node n) {
  // Find & delete the edges connected to this node
  for (int j=0; j<g.numEdges; j++) {
    if ((g.edges[j].startNode == n) || (g.edges[j].endNode == n)) {
      // Put the last edge in the place of this deleted edge
      g.edges[j] = g.edges[g.numEdges-1];
      g.numEdges--;
      j--;   // required so that we check the same index next time
    }
  }
  // Now find and delete the Node
  for (int i=0; i<g.numNodes; i++) {
    if (g.nodes[i] == n) {
      // Put the last node in the place of this deleted node
      g.nodes[i] = g.nodes[g.numNodes-1];
      g.numNodes--;
      return;
    }
  }
}
```

Notice that to delete something from an array we need to replace it with a different value. Rather than leave gaps, the above code takes the last edge (or node ... depending on the array) and places it in the array at the position of the one deleted. Then the number of elements in the array is actually smaller, so we decrease **numEdges** (or **numNodes** ... depending on the array). The **j--** is required to ensure that the next time through the loop ... we check this same location that we removed the edge (or node) from ... since we placed the last edge (or node) there and it too needs to be checked.

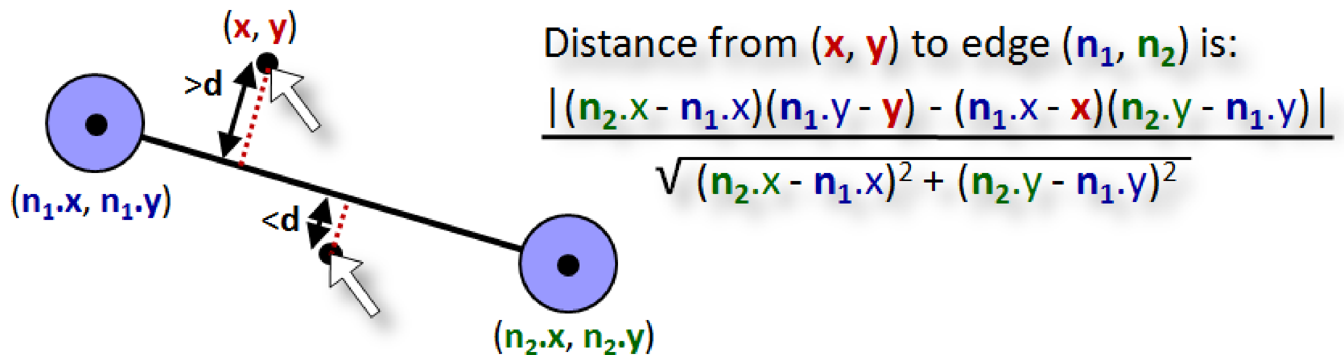
The code should now work, allowing all selected nodes to be deleted.

Selecting Edges:

We will also want to select, move and delete edges. We can add a **boolean** to the edge data structure to allow this:

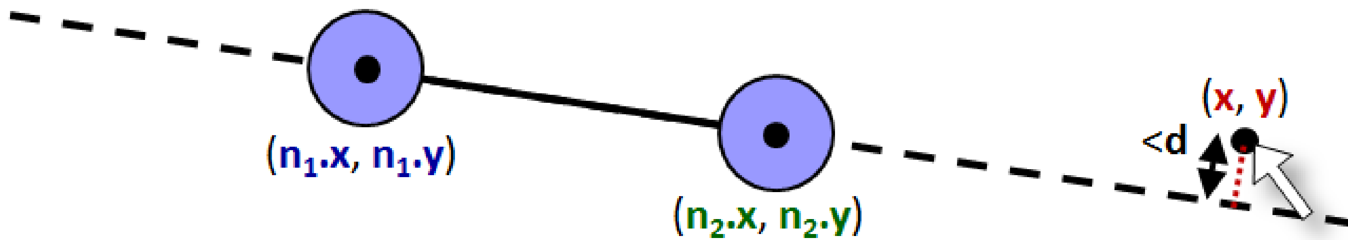
```
class Edge {
  Node       startNode, endNode;
  boolean    selected;

  Edge(Node start, Node end) {
    startNode = start;
    endNode = end;
    selected = false;
  }
}
```

How does the user select an edge ? Likely, by clicking on or near it. We can accomplish this by determining the distance between the point clicked at and the edge itself. If the distance is smaller than some pre-decided value (e.g., 5 pixels) then we can assume that this edge was just clicked on... otherwise we can assume that the edge was not clicked on. The equation to find the distance from a point to an edge is indicated below:



$$\frac{|(n_2.x - n_1.x)(n_1.y - y) - (n_1.x - x)(n_2.y - n_1.y)|}{\sqrt{(n_2.x - n_1.x)^2 + (n_2.y - n_1.y)^2}}$$

However, the above equation actually computes the distance from (**x,y**) to the **line** that passes through the two edge nodes. So, if we click anywhere close to that line, we will be a small distance value and we will think that the edge was selected:



Certainly, we do not want such an (**x,y**) point to be considered as "close to" the edge. We can avoid this problem situation by examining the x-coordinate of the point that the user clicked on. The x-coordinate must be greater that the left node's x-coordinate and smaller than the right node's x-coordinate.

```
IF (distance < 3) THEN {
        IF ((x > n1.x) AND (x < n2.x)) OR ((x > n2.x) AND (x < n1.x)) THEN
                this edge has been selected
}
```

Can you foresee any further problems with the algorithm ? What if the edge is vertical ? The above checking will never select the edge ! Instead, if the edge is vertical, we should compare the y-coordinates. In fact, if the line is "more horizontal" we should check the x-coordinates and if it is "more vertical" we should check the y-coordinates. To determine if a line segment is more vertical or horizontal, we can compare the difference in x and the difference in y. Here is the code:

```
IF (distance < 3) THEN {
        xDiff ← abs(n2.x - n1.x)
        yDiff ← abs(n2.y - n1.y)
        IF (xDiff > yDiff) THEN
                IF ((x > n1.x) AND (x < n2.x)) OR ((x > n2.x) AND (x < n1.x)) THEN
                        this edge has been selected
        OTHERWISE
                IF ((y > n1.y) AND (y < n2.y)) OR ((y > n2.y) AND (y < n1.y)) THEN
                        this edge has been selected
}
```

Now that we have the algorithm that we need, it would make sense to create an **edgeAt()** function to find the edge that was selected, since we did something similar for **nodeAt()**.

Here is the code:

```
Edge edgeAt(Graph g, int x, int y) {
  for (int i=0; i<g.numEdges; i++) {
    Edge e = g.edges[i];
    Node n1 = e.startNode;
    Node n2 = e.endNode;
    int xDiff = n2.x - n1.x;
    int yDiff = n2.y - n1.y;
    float distance = abs(xDiff*(n1.y-y)-(n1.x-x)*yDiff) /
                     sqrt(xDiff*xDiff + yDiff*yDiff);
    if (distance <= 5) {
      if (abs(xDiff) > abs(yDiff)) {
        if (((x < n1.x) && (x > n2.x)) || ((x > n1.x) && (x < n2.x)))
          return e;
      }
      else
        if (((y < n1.y) && (y > n2.y)) || ((y > n1.y) && (y < n2.y)))
          return e;
    }
  }
  return null;
}
```

Now that we have such a function, we can modify the **mouseClicked** event handler to allow edges to be selected:

```
void mouseClicked() {
  if (mouseEvent.getClickCount() == 2) {
    Node n = nodeAt(graph, mouseX, mouseY);
    if (n != null)
      n.selected = !n.selected;
    else {
      Edge e = edgeAt(graph, mouseX, mouseY);
      if (e != null)
        e.selected = !e.selected;
      else
        addNode(graph, new Node(mouseX, mouseY, ""));
    }
  }
}
```

Notice now that we first check to see if a node was selected and only if a node was not selected will we then check to see if an edge was selected. Of course, we will want to show selected edges as red, so we need to modify the **draw()** procedure:

```
if (e.selected) {
  stroke(255,0,0);   // red edge
  strokeWeight(5);   // thicker line
}
else {
  stroke(0,0,0);     // black edge
  strokeWeight(1);   // thin line
}
```

Our program now allows edges to be selected.

Deleting Edges:

We will want to allow edges to be deleted as well. To do this, we should allow the user to select any edges and press delete. To delete an edge, we simply need to remove it from the array. We can write a **deleteEdge()** procedure to do this as follows:

```
void deleteEdge(Graph g, Edge e) {
  for (int i=0; i<g.numEdges; i++) {
    if (g.edges[i] == e) {
      // Put the last edge in the place of this deleted edge
      g.edges[i] = g.edges[g.numEdges-1];
      g.numEdges--;
      return;
    }
  }
}
```

Then, in the **keyPressed** event handler we simply remove any selected edges by adding the following code:

```
for (int i=0; i<graph.numEdges; i++) {
  if (graph.edges[i].selected) {
    deleteEdge(graph, graph.edges[i]);
    i--;
  }
}
```

Moving Edges:

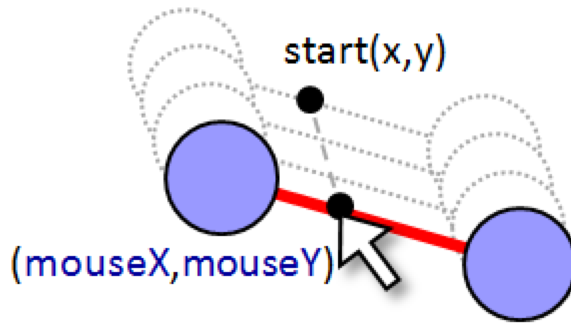
It would be nice to allow an edge to be grabbed and dragged just like a node is grabbed and dragged. To do this, we will need to determine whether or not the user has pressed the mouse on an edge and then "remember" that edge so that it can be dragged later. For nodes, we used a **dragNode** variable. For edges we can create a similar **dragEdge** variable:

```
Edge     dragEdge;
```

We can set this to **null** in the **setup()** procedure and then in the **mousePressed** event handler we can set it as follows:

```
dragEdge = edgeAt(graph, mouseX, mouseY);
```

At this point, we will know which edge has been selected for dragging. All that is left to do is to actually move the edge when the user drags the mouse. However, how do we actually move the edge ? It should move by the amount that the mouse moved. We can calculate the amount that the mouse moved by remembering the "start" point that was initially clicked on and then finding the change in **x** and **y** to the current mouse location:



So, in addition to the edge that has been selected, we will need to remember the point that was clicked at so that we can determine the change from that point to the new mouse location in order to adjust the edge's node locations. We can create two new variables at the top of the program to store the mouse location at which the user initially pressed:

```
int      dragX, dragY;
```

We can then add this to the **mousePressed** handler set the values:

```
dragX = mouseX;
dragY = mouseY;
```

Then in the **mouseDragged** event handler we can now move the edge, provided that one was selected by the user by appending this code:

```
if (dragEdge != null)
    if (dragEdge.selected) {
      int x = mouseX - dragX;
      int y = mouseY - dragY;
      dragEdge.startNode.x = dragEdge.startNode.x + x;
      dragEdge.startNode.y = dragEdge.startNode.y + y;
      dragEdge.endNode.x = dragEdge.endNode.x + x;
      dragEdge.endNode.y = dragEdge.endNode.y + y;
      dragX = mouseX;
      dragY = mouseY;
    }
```

Moving Multiple Nodes:

When we moved the edge in the above code, we actually did this by moving two nodes. In fact, we can adjust our node-dragging code so that it allows multiple selected nodes to be moved around at the same time. We just need to offset the nodes by an amount equal to the difference between the current mouse location (**mouseX**, **mouseY**) and the (**dragX**, **dragY**) location ... as when we moved the edge.

So, instead of doing this in the **mouseDragged** event handler:

```
dragNode.x = mouseX;  dragNode.y = mouseY;
```

we can do the following to move ALL the selected nodes:

```
for(int i=0; i<graph.numNodes; i++) {
  if (graph.nodes[i].selected) {
    graph.nodes[i].x = graph.nodes[i].x + x;
    graph.nodes[i].y = graph.nodes[i].y + y;
  }
}
```

Now, for example, if we selected all the nodes in the graph, we could translate the whole graph with one drag operation !

Other Fun Features:

If you want to have more fun, try implementing these features on your own:

- Allow all selected edges and nodes to be moved by dragging a selected edge.
- Press <CNTRL><A> to select all nodes and edges and <CNTRL><U> to unselect them.
- Right-click the mouse on a Node and prompt the user for a label to put on that node.
- Scale the entire graph up or down by holding the <SHIFT> key while pressing the mouse on an empty spot on the window and then dragging the mouse up or down to enlarge or shrink the graph.
- Press <CNTRL><D> to duplicate all selected nodes and edges and have the new portion of the graph appear a little below and to the right of the original nodes/edges.