# Chapter 10

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.
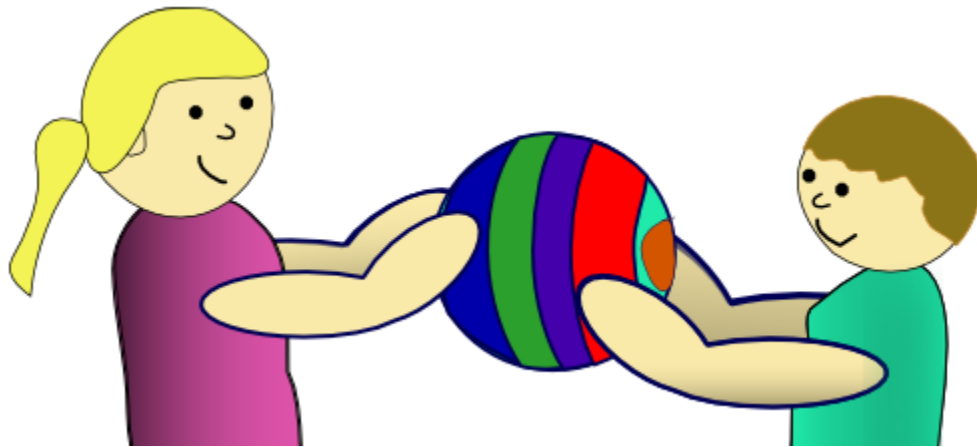
10.1 Sharing Data Can Be Useful

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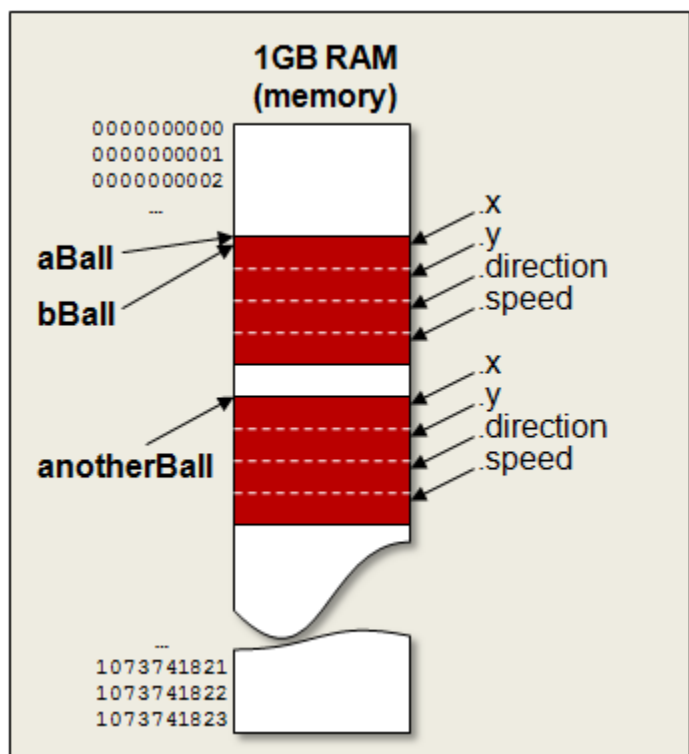
Of course, sharing does have some disadvantages.   For example, if someone borrows the car and brings it back with no gasoline, then it affects the next person who will use the car.   Also, the seat and mirrors may all need to be adjusted for the next driver. Worse yet, if one person in the family crashes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

```java
import java.awt.Color;

public class Beacon {
   public static final int  RADIUS = 15;    // The radius of the beacon (in pixels)

   int       x, y;     // location of the beacon at any time
   Color     col;      // color of the beacon

   public Beacon(int bx, int by) {
      x = bx;
      y = by;
      col = new Color(55 + (int)(Math.random()*200),
                      55 + (int)(Math.random()*200),
                      55 + (int)(Math.random()*200));
   }
}
```

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

```java
public class Robot {
   public static final int  RADIUS = 5;    // The radius of the robot (in pixels)

   int      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

   public Robot(int rx, int ry) {
      x = rx;  y = ry;
      direction = (float)(Math.random()*Math.PI*2);   // a random direction
      speed = 3 + (float)(Math.random()*4);           // a random speed from 3 to 6
      beacon = null;                                   // not set yet
   }
}
```

Notice that each robot stores their own **Beacon** object (i.e., their own place to head towards).

Here is a **JPanel** that we will use to display the robots and their beacons. It is very similar to the **BallPanel** class in our **BallSimulation**. Do not worry about the actual drawing code.

```java
import java.awt.*;
import javax.swing.*;

// This code is responsible for displaying the Robots in their environment
public class RobotEnvironmentPanel extends JPanel {
    public static final int WIDTH = 800;
    public static final int HEIGHT = 600;

    private static Robot[]  robots;

    public RobotEnvironmentPanel(Robot[] rArray) {
        robots = rArray;
        setPreferredSize(new Dimension(WIDTH, HEIGHT));
    }

    // Display the image
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        // Display the beacons first
        for (int i=0; i<robots.length; i++) {
            g.setColor(robots[i].beacon.col);    // Get the beacon from the robot
            g.fillOval(robots[i].beacon.x-Beacon.RADIUS,
                       robots[i].beacon.y-Beacon.RADIUS,
                       Beacon.RADIUS*2,
                       Beacon.RADIUS*2);
            g.setColor(Color.BLACK);
            g.drawOval(robots[i].beacon.x-Beacon.RADIUS,
                       robots[i].beacon.y-Beacon.RADIUS,
                       Beacon.RADIUS*2,
                       Beacon.RADIUS*2);
        }
        // Now display the robots
        for (int i=0; i<robots.length; i++) {
            // Display the robot
            g.setColor(robots[i].beacon.col);  // Robot has same color as beacon
            g.fillOval(robots[i].x-Robot.RADIUS,
                       robots[i].y-Robot.RADIUS,
                       Robot.RADIUS*2,
                       Robot.RADIUS*2);
            g.setColor(Color.BLACK);
            g.drawOval(robots[i].x-Robot.RADIUS,
                       robots[i].y-Robot.RADIUS,
                       Robot.RADIUS*2,
                       Robot.RADIUS*2);
        }
    }
}
```

Notice how the **Beacon.RADIUS** and **Robot.RADIUS** constants are used in the drawing.
Also notice that the robot is drawn according to the color of the beacon that it is following.

Here is the main Simulation code:

```java
import java.awt.*;      // Needed for window and graphics (explained in COMP1406)
import javax.swing.*;   // Needed for window and graphics (explained in COMP1406)

// This application simulates robots attracted to beacons in a window
public class RobotSimulation {
    // This variable stores the panel that displays the robots and beacons
    // (This is a topic discussed in COMP1406 course)
    public static RobotEnvironmentPanel   envPanel;
    public static Robot[]                  robots;      // array to hold robots

    public static void startSimulation() {
        while(true) {
            for (int i=0; i<robots.length; i++)
                move(robots[i]);
            envPanel.repaint();
            try { Thread.sleep(10); } catch(Exception e){};
        }
    }

    // Code for moving a robot towards its beacon
    public static void move(Robot r) {
        /* ... code has been left out ... described later ... */
    }

    // Create a window, add beacons and robots, then start simulating
    public static void main(String args[]) {
        // Make some robots with unique beacons
        robots = new Robot[10];
        for (int i=0; i<robots.length; i++) {
            robots[i] = new Robot(RobotEnvironmentPanel.WIDTH/2,
                                  RobotEnvironmentPanel.HEIGHT/2);
            robots[i].beacon = new Beacon((int)(Math.random() *
                                                    RobotEnvironmentPanel.WIDTH),
                                          (int)(Math.random() *
                                                    RobotEnvironmentPanel.HEIGHT));
        }

        JFrame frame = new JFrame("Robot Simulation");
        frame.add(envPanel = new RobotEnvironmentPanel(robots));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);

        startSimulation();
    }
}
```

Notice in the **main()** method that an array of **10** robots is created and each robot starts at the center of the window.   Each robot is given a new beacon whose location is chosen randomly. The simulation itself is simply an infinite loop in which each robot is moved forward a bit one at a time and then the window is repainted and a small **10ms** delay is used to slow down the simulation.

The code for moving the robot has been left out, but will be described later. For now, assume though that it steers the robot towards its beacon location.

Consider drawing the array of robots and their beacons as follows ... notice how each robot has its own **Beacon** object, allowing them all to head in different directions:
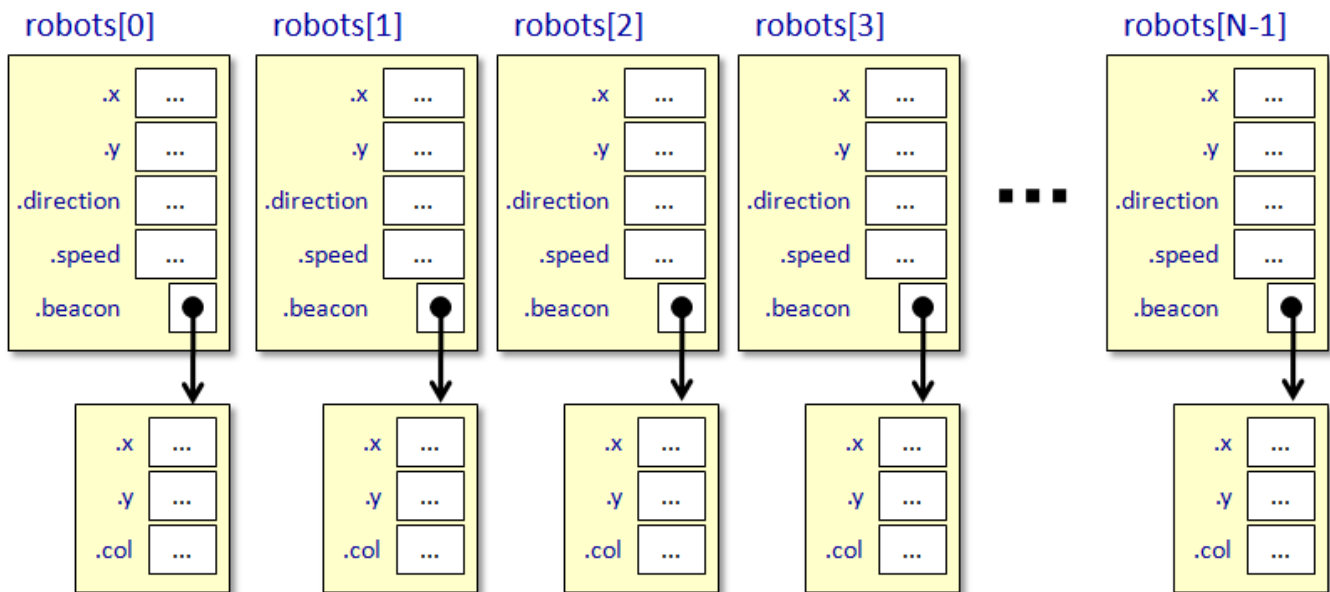
In our code, each robot has its own unique beacon to head towards.   But now, we will adjust the code to allow multiple robots to head towards the same beacon, thereby causing a swarm-like simulation.   We will alter the code so that instead of having a unique beacon for each robot, we will generate 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:
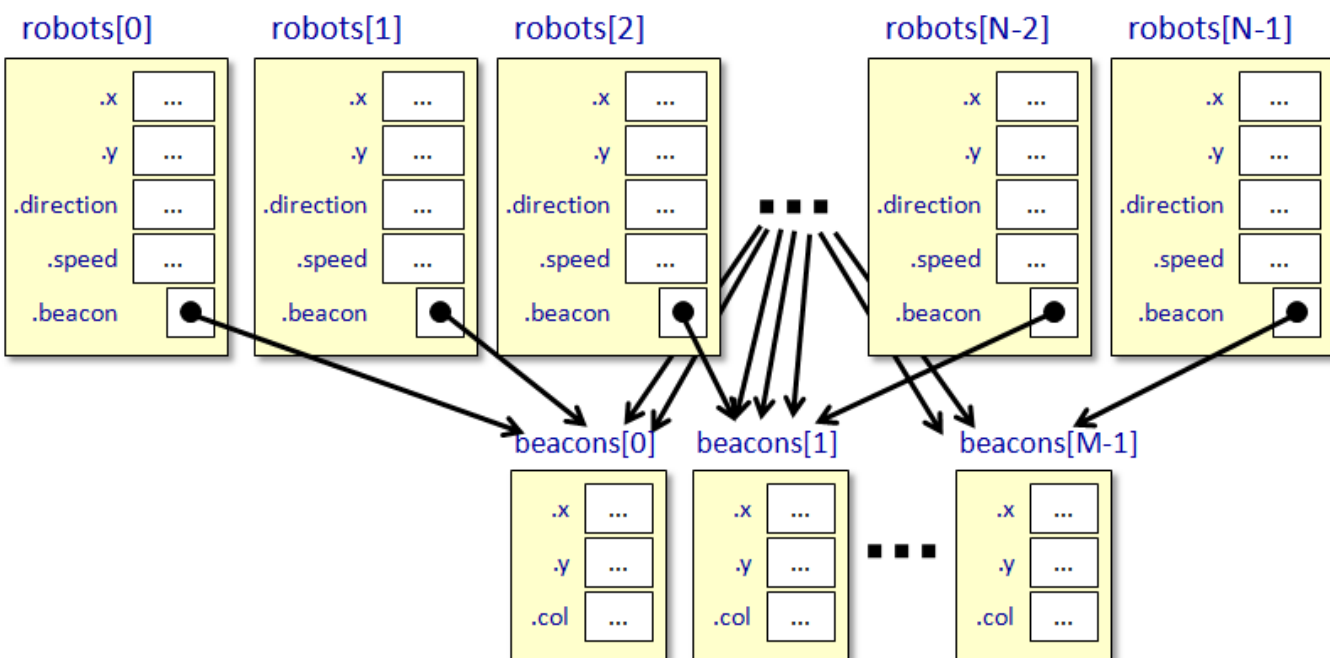
There will be less **Beacon** objects created since multiple robots will share the same one.   Is there any advantage to having less/shared **Beacons** ?

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.   Consider having just 5 beacon locations, but over 100 robots who all head to one of the 5 beacons.   If a 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 used unique beacon objects with many overlapping at the same location and then stored a unique (i.e., non-shared) beacon within each robot, it would be harder to simulate.   When a beacon changes locations, we would need to find all robot's that have beacons that share that beacon location 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 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.   First, we will produce a new **RoboEnvironmentPanel** that will allow us to pass in beacons for display purposes:

```java
import java.awt.*;
import javax.swing.*;

// This code is responsible for displaying the Robots in their environment
public class RobotEnvironmentPanel2 extends JPanel {
    public static final int WIDTH = 800;
    public static final int HEIGHT = 600;

    private static Robot[]  robots;
    private static Beacon[] beacons;
```

```java
    public RobotEnvironmentPanel2(Robot[] rArray, Beacon[] bArray) {
        robots = rArray;
        beacons = bArray;
        setPreferredSize(new Dimension(WIDTH, HEIGHT));
    }

    // Display the image
    public void paintComponent(Graphics g) {
        super.paintComponent(g);

        // Display the beacons first
        for (int i=0; i<beacons.length; i++) {
            g.setColor(beacons[i].col);
            g.fillOval(beacons[i].x-Beacon.RADIUS,
                       beacons[i].y-Beacon.RADIUS,
                       Beacon.RADIUS*2,
                       Beacon.RADIUS*2);
            g.setColor(Color.BLACK);
            g.drawOval(beacons[i].x-Beacon.RADIUS,
                       beacons[i].y-Beacon.RADIUS,
                       Beacon.RADIUS*2,
                       Beacon.RADIUS*2);
        }
        // Now display the robots
        for (int i=0; i<robots.length; i++) {
            g.setColor(robots[i].beacon.col);
            g.fillOval(robots[i].x-Robot.RADIUS,
                       robots[i].y-Robot.RADIUS,
                       Robot.RADIUS*2,
                       Robot.RADIUS*2);
            g.setColor(Color.BLACK);
            g.drawOval(robots[i].x-Robot.RADIUS,
                       robots[i].y-Robot.RADIUS,
                       Robot.RADIUS*2,
                       Robot.RADIUS*2);
        }
    }
}
```

Now, for the main **RobotSimulation** program, we will simply alter the main method to share the beacons:

```java
import java.awt.*;     // Needed for window and graphics (explained in COMP1406)
import javax.swing.*;  // Needed for window and graphics (explained in COMP1406)

// This application simulates robots attracted to beacons in a window
public class RobotSimulation2 {
    // This variable stores the panel that displays the robots and beacons
    // (This is a topic discussed in COMP1406 course)
    public static RobotEnvironmentPanel   envPanel;

    public static Robot[]    robots;                // an array to hold the robots
    public static Beacon[]    beacons;              // an array to hold the beacons

    public static void startSimulation() {
        /* ... same code as before ... */
    }
```

```
    public static void move(Robot r) {
        /* ... same code as before ... */
    }

    // Create a window, add beacons and robots, then start simulating
    public static void main(String args[]) {
        // Make some beacons
        beacons = new Beacon[5];
        for (int i=0; i<beacons.length; i++) {
            beacons[i] = new Beacon((int)(Math.random() *
                                            RobotEnvironmentPanel2.WIDTH),
                                    (int)(Math.random() *
                                            RobotEnvironmentPanel2.HEIGHT));
        }
        // Make some robots with shared beacons
        robots = new Robot[1000];
        for (int i=0; i<robots.length; i++) {
            robots[i] = new Robot(RobotEnvironmentPanel2.WIDTH/2,
                            RobotEnvironmentPanel2.HEIGHT/2);
            robots[i].beacon = beacons[(int)(Math.random()*beacons.length)];
        }

        JFrame frame = new JFrame("Robot Simulation");
        frame.add(envPanel = new RobotEnvironmentPanel2(robots));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack();
        frame.setVisible(true);

        startSimulation();
    }
}
```


Notice now how each robot is assigned one of the 5 beacons at random. With 1000 robots, you can see how they are grouped into 5 swarms now because roughly 200 are assigned to each beacon.

For added enjoyment, we can add the following to allow beacons to be grabbed and moved around on the screen. Do not worry about understanding this code ... as it will be discussed in another course:

```
import java.awt.*;      // Needed for window and graphics (explained in COMP1406)
import javax.swing.*;   // Needed for window and graphics (explained in COMP1406)

// This application simulates robots attracted to beacons in a window
public class RobotSimulation3 {
    // This variable stores the panel that displays the robots and beacons
    // (This is a topic discussed in COMP1406 course)
    public static RobotEnvironmentPanel   envPanel;

    public static Robot[]     robots;           // an array to hold the robots
    public static Beacon[]    beacons;          // an array to hold the beacons
    public static Beacon      grabbed;          // Beacon that user just grabbed
```

```java
    public static void startSimulation() {
        /* ... same code as before ... */
    }
    public static void move(Robot r) {
        /* ... same code as before ... */
    }

    // Handle the pressing of the mouse button
    public static void handleMousePress(int mouseX, int mouseY) {
        for (int i=0; i<beacons.length; i++) {
            if (java.awt.geom.Point2D.distance(beacons[i].x, beacons[i].y,
                                               mouseX, mouseY) < Beacon.RADIUS) {
                grabbed = beacons[i];
                return;
            }
        }
    }

    // Handle the releasing of the mouse button
    public static void handleMouseRelease(int mouseX, int mouseY) {
        grabbed = null;
    }

    // Handle the dragging of the beacon
    public static void handleMouseDragged(int mouseX, int mouseY) {
        if (grabbed != null) {
            grabbed.x = mouseX;
            grabbed.y = mouseY;
        }
    }

    // Create a window, add beacons and robots, then start simulating
    public static void main(String args[]) {
        // Make some beacons
        beacons = new Beacon[5];
        for (int i=0; i<beacons.length; i++) {
            beacons[i] = new Beacon((int)(Math.random() *
                                            RobotEnvironmentPanel2.WIDTH),
                                    (int)(Math.random() *
                                            RobotEnvironmentPanel2.HEIGHT));
        }
        // Make some robots with shared beacons
        robots = new Robot[1000];
        for (int i=0; i<robots.length; i++) {
            robots[i] = new Robot(RobotEnvironmentPanel2.WIDTH/2,
                            RobotEnvironmentPanel2.HEIGHT/2);
            robots[i].beacon = beacons[(int)(Math.random()*beacons.length)];
        }

        JFrame frame = new JFrame("Robot Simulation");
        frame.add(envPanel = new RobotEnvironmentPanel2(robots));

        // Plug in the methods that allow handling of the mouse events
        // Do not worry about this, event handling will be discussed in
        // a later course
        envPanel.addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent e) {
                        handleMousePress(e.getX(), e.getY()); }
            public void mouseReleased(MouseEvent e) {
                        handleMouseRelease(e.getX(), e.getY());
```

```
    }});
    envPanel.addMouseMotionListener(new MouseMotionAdapter() {
        public void mouseDragged(MouseEvent e) {
                            handleMouseDragged(e.getX(), e.getY());
    }});

    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.pack();
    frame.setVisible(true);

    startSimulation();
  }
}
```

As a 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:

```
public static void move(Robot r) {
      r.x = r.x + (int)(r.speed * Math.cos(r.direction));
      r.y = r.y + (int)(r.speed * Math.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:
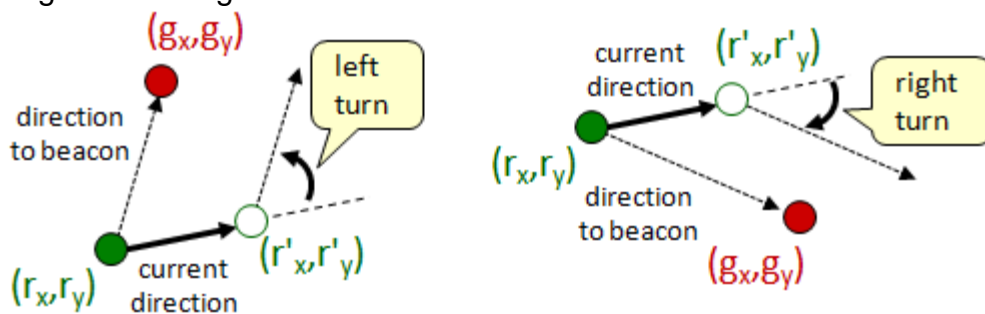

Above, (r_x, r_y) is the robot's current location and (r'_x, r'_y) would be the robot's next location if it were to travel in its current direction without turning (for this next location we could simply use the new value for **r.x** and **r.y** that we computed in our code above).

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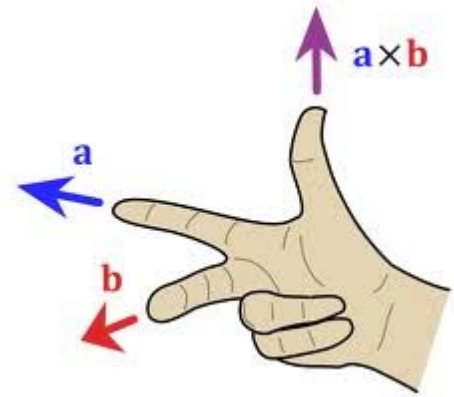
If the cross product is positive, this indicates a left turn. If negative, then a right turn. If the cross product is zero, then there is no turn (i.e., the two vectors form an angle of 180°).
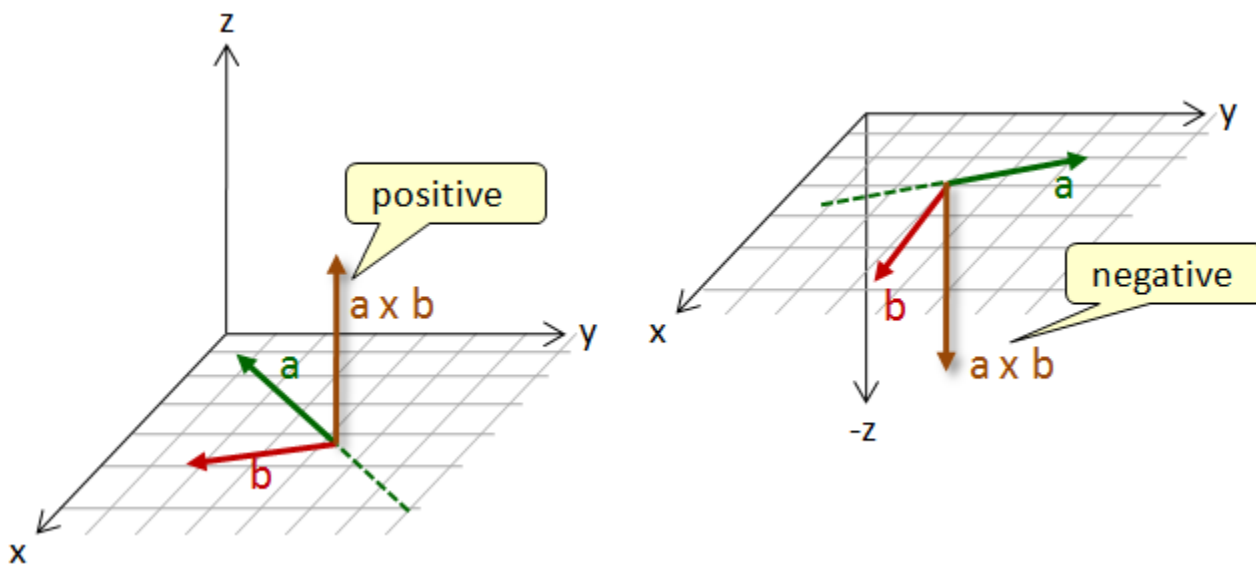
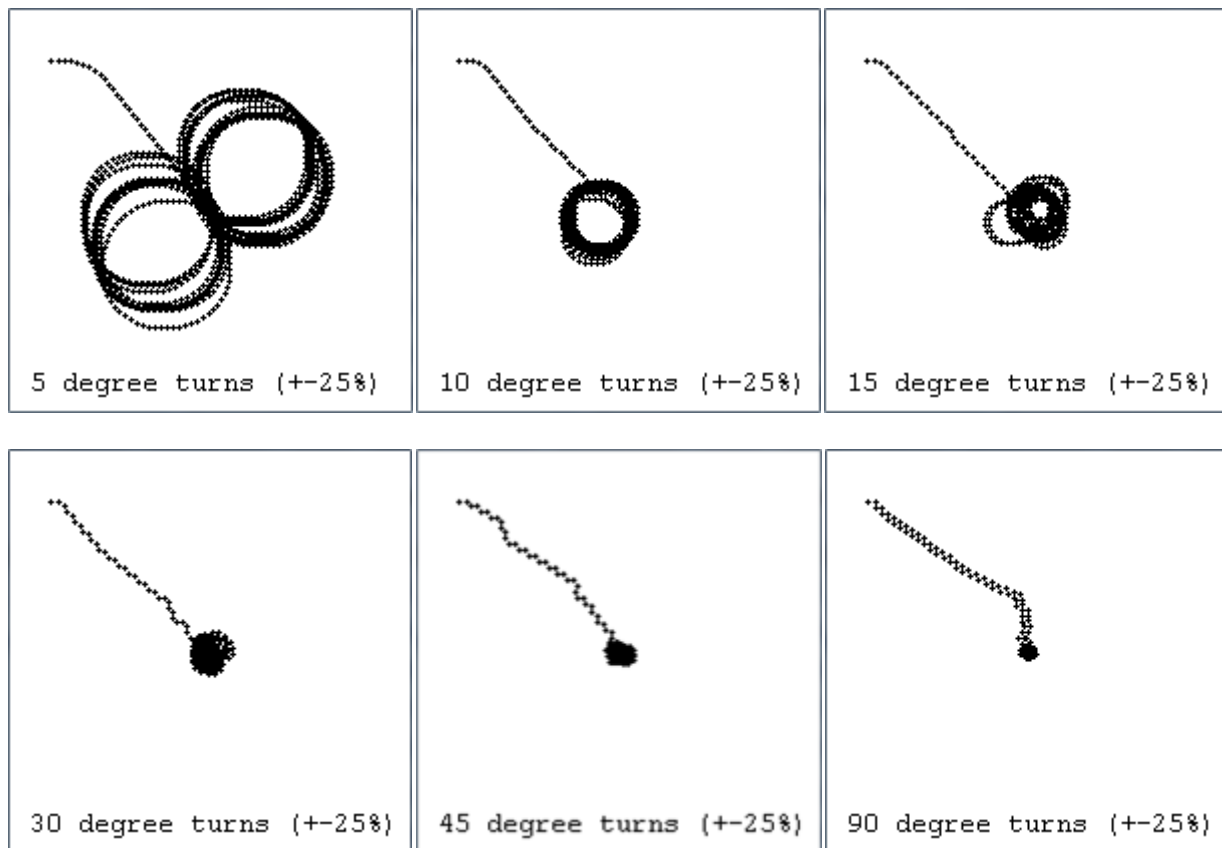The cross product can be computed as follows:

$$crossProduct = (r'_x - r_x)(g_y - r_y) - (r'_y - r_y)(g_x - r_x)$$

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):

- **318** -

5 degree turns (+-25%)   10 degree turns (+-25%)   15 degree turns (+-25%)

30 degree turns (+-25%)   45 degree turns (+-25%)   90 degree turns (+-25%)

As a result, here is the final move method:

```java
public static void move(Robot r) {

    float nextX, nextY, crossProduct;

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

    crossProduct = (nextX - r.x)*(r.beacon.y - r.y)-
                   (nextY - r.y)*(r.beacon.x - r.x);

    r.x = (int)nextX;
    r.y = (int)nextY;

    if (crossProduct < 0)
        r.direction -= (Robot.TURN_ANGLE + Math.random()*
                        Robot.TURN_ANGLE/4 - (Robot.TURN_ANGLE/8))/180*Math.PI;
    else
        r.direction += (Robot.TURN_ANGLE + Math.random()*
                        Robot.TURN_ANGLE/4 - (Robot.TURN_ANGLE/8))/180*Math.PI;
}
```

Here the **Robot.TURN_ANGLE** is a static constant in the **Robot** class:

```java
public static final int  TURN_ANGLE = 15;
```

10.2 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 is 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:
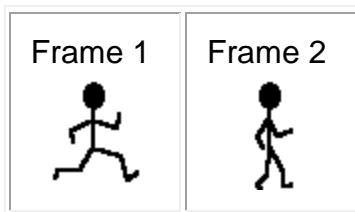
The animation, however, would have the undesirable characteristic of being very "jumpy", not very smooth.   The problem is that there is no smooth transition between the frames. We can make a big improvement just by introducing one more picture and duplicating the 2nd frame twice to produce a 4 frame sequence:
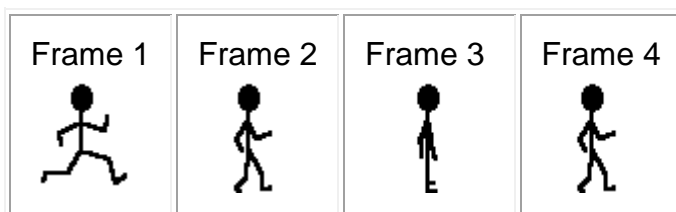
This animation would appear much smoother, but if we display the frames at the same rate, the person would appear to walk much slower.   Assuming that we display 1 frame every 1/4 second, the 2-frame case would take 1/2 second to complete a cycle while in the 4-frame case, would take a full second.   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 in the folder that contains our code.

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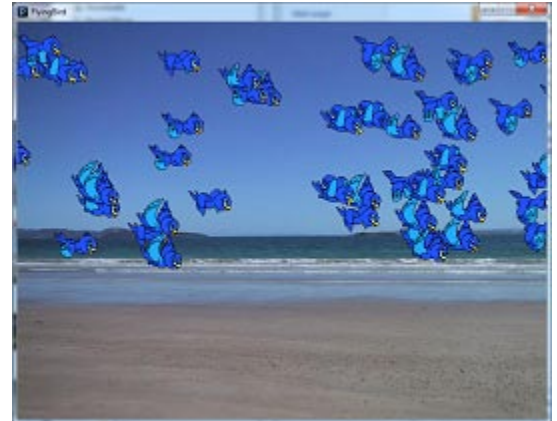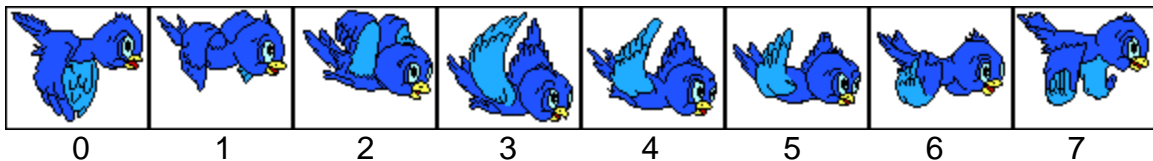
We will need to maintain information for each bird in regards to its location on the screen, its speed and perhaps which frame is being shown at any time.   Here is how we can define such a **Bird** data structure. Define the **Bird** class as follows:

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

    public Bird(float bx, float by) {
        x = bx;
        y = by;

        // Choose a random speed from 5 to 15 pixels/frame and random picture
        speed = (float)(Math.random()*10 + 5);
        currentFrame = (int)(Math.random()*8);

        // load up all 8 pictures for this bird
        images = new java.awt.Image[8];              // array to hold 8 pictures
        for (int i=1; i<=8; i++)
            images[i-1] = java.awt.Toolkit.getDefaultToolkit().
                                        createImage("Birdx" + i + ".gif");
    }
}
```

Here is the **BirdPanel** that will be used to display the birds with a beach background image. You need not worry about understanding this.   The code is quite similar to the code for drawing the fire simulation, the balls and the robots.

```java
import java.awt.*;
import javax.swing.*;

public class BirdPanel extends JPanel {
    public static final int WIDTH = 800;
    public static final int HEIGHT = 600;

    private static Image   anImage;
    private static Bird[]  birds;

    public BirdPanel(Bird[] b) {
        birds = b;
        anImage = Toolkit.getDefaultToolkit().createImage("beach.jpg");;
        setPreferredSize(new Dimension(WIDTH, HEIGHT));
    }

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawImage(anImage, 0, 0, null);
        for (int i=0; i<birds.length; i++)
            g.drawImage(birds[i].images[birds[i].currentFrame],
                        (int)birds[i].x, (int)birds[i].y, null);
    }
}
```

For the simulation, we can create an array of **Bird** objects and have them simply fly forwards.
The **startSimulation()** method is similar to our other simulation programs.

```java
import java.awt.*;
import javax.swing.*;

// This application simulates birds flying in a window
public class BirdSimulation {
    public static BirdPanel  birdPanel;  // Panel to display the birds
    public static Bird[]     birds;      // an array to hold the birds

    public static void startSimulation() {
        while(true) {
            for (int i=0; i<birds.length; i++)
                move(birds[i]);
            birdPanel.repaint();
            try{ Thread.sleep(60); } catch(Exception e){};
        }
    }

    // Code for moving a bird forwards
    public static void move(Bird b) {
        // ... Discussed soon ... //
    }

    // Create a window, add birds, then start simulating
    public static void main(String args[]) {
        // Make some birds
        birds = new Bird[2000];        // an array to hold the birds
        for (int i=0; i<birds.length; i++) {
            birds[i] = new Bird((float)(Math.random()*BirdPanel.WIDTH),
                            (float)(Math.random()*BirdPanel.HEIGHT/2));
        }
```

```
        JFrame frame = new JFrame("Bird Simulation");
        frame.add(birdPanel = new BirdPanel(birds));
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.pack(); // Makes size according to panel's preference
        frame.setVisible(true);

        startSimulation();
    }
}
```

To animate the birds, we must now do two things: (1) Have them vary their images, and (2) have them move horizontally.

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. Here is the code:

```
// Code for moving a bird forwards
public static void move(Bird b) {
    b.x = b.x + b.speed; // Move the bird 10 pixels forward

    if (b.x > BirdPanel.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 **2500** birds. Depending on your computer's memory ... the code may or may not run.   My IDE crashed with an error. What is the problem ?

The current code loads **8** images for each bird. Therefore, **2500** birds would require (**2500** x **8** = **20,000**) 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 make the images static instead of having them as instance variables.   That way, the single set of images is stored and shared by all Bird objects.   Here is the code:

```java
public class Bird {
    // Frames/pictures of the bird in various poses
    public static final java.awt.Image[]   images = new java.awt.Image[8];

    float    x, y;          // bird's coordinate on the screen
    float    speed;         // bird's speed in pixel movements per frame
    int      currentFrame;  // current frame of the bird

    public Bird(float bx, float by) {
        x = bx;
        y = by;

        // Choose a random speed from 5 to 15 pixels/frame and random picture
        speed = (float)(Math.random()*10 + 5);
        currentFrame = (int)(Math.random()*8);

        // load up all 8 pictures for this bird
        for (int i=1; i<=8; i++)
            images[i-1] = java.awt.Toolkit.getDefaultToolkit().
                                        createImage("Birdx" + i + ".gif");
    }
}
```

Also, in our **BirdPanel**, we need to change this:

```java
    g.drawImage(birds[i].images[birds[i].currentFrame],
                (int)birds[i].x, (int)birds[i].y, null);
```

to this:

```java
    g.drawImage(Bird.images[birds[i].currentFrame],
                (int)birds[i].x, (int)birds[i].y, null);
```

Now we can add 2500 birds no problem.   In fact, we can add 50,000!

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...

10.3 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 of people:

```java
public static Person[] makeCopy(Person[] aList) {
      Person[] theCopy = new Person[aList.length];

      for (int i=0; i<aList.length; i++)
            theCopy[i] = aList[i];

      return theCopy;
}
```

The code produces a new array containing the people 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:

```java
Person[] original = new Person[3];
original[0] = new Person("Hank", "Urchif", 21, 5.8, false);
original[1] = new Person("Holly", "Day", 32, 5.9, false);
original[2] = new Person("Bobby", "Pins", 87, 6.2, true);

Person[] copy = makeCopy(original);
```

Here is what we accomplish with this code …
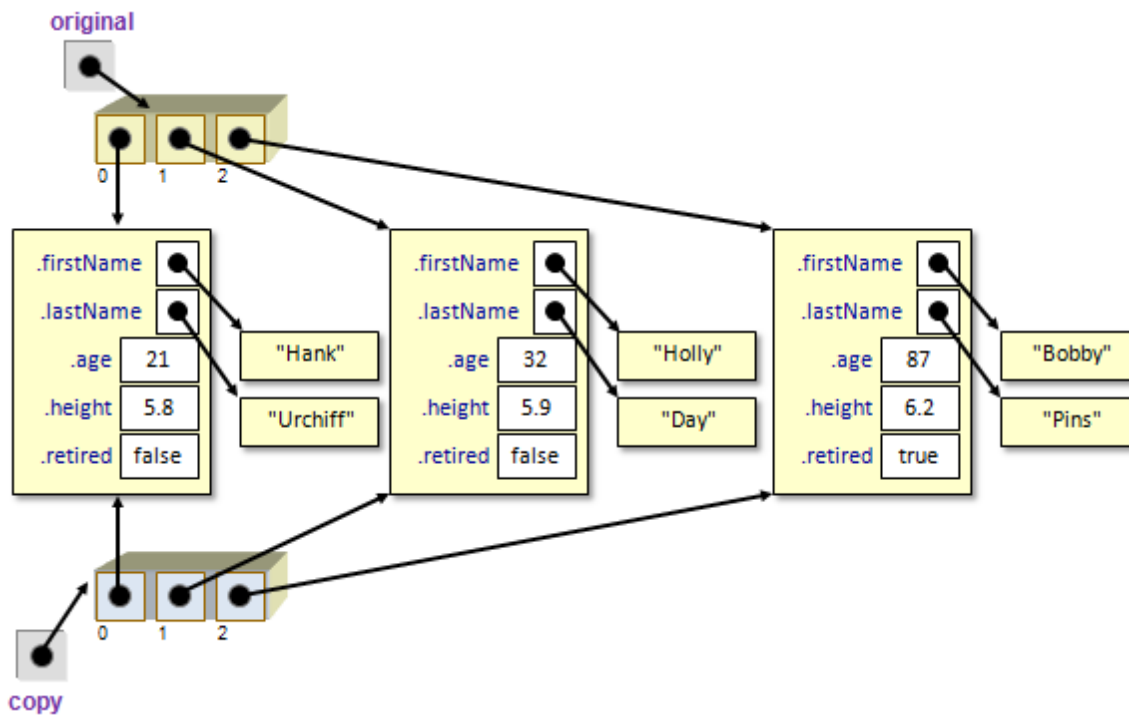
Notice that when we add the **Person** objects to the copy, they are actually shared between the two arrays. This is known as a ***shallow copy***. The advantage is that we do not need to use up additional memory space to store duplicate information for the copy.   That is, we do not need to store the 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:

```
Person[] original = new Person[3];
original[0] = new Person("Hank", "Urchif", 21, 5.8, false);
original[1] = new Person("Holly", "Day", 32, 5.9, false);
original[2] = new Person("Bobby", "Pins", 87, 6.2, true);

Person[] copy = makeCopy(original);

copy[0].firstName = "Steve";
copy[2] = new Person("April", "Rain", 15, 4.7, false);
copy[2].firstName = "Acid";
```
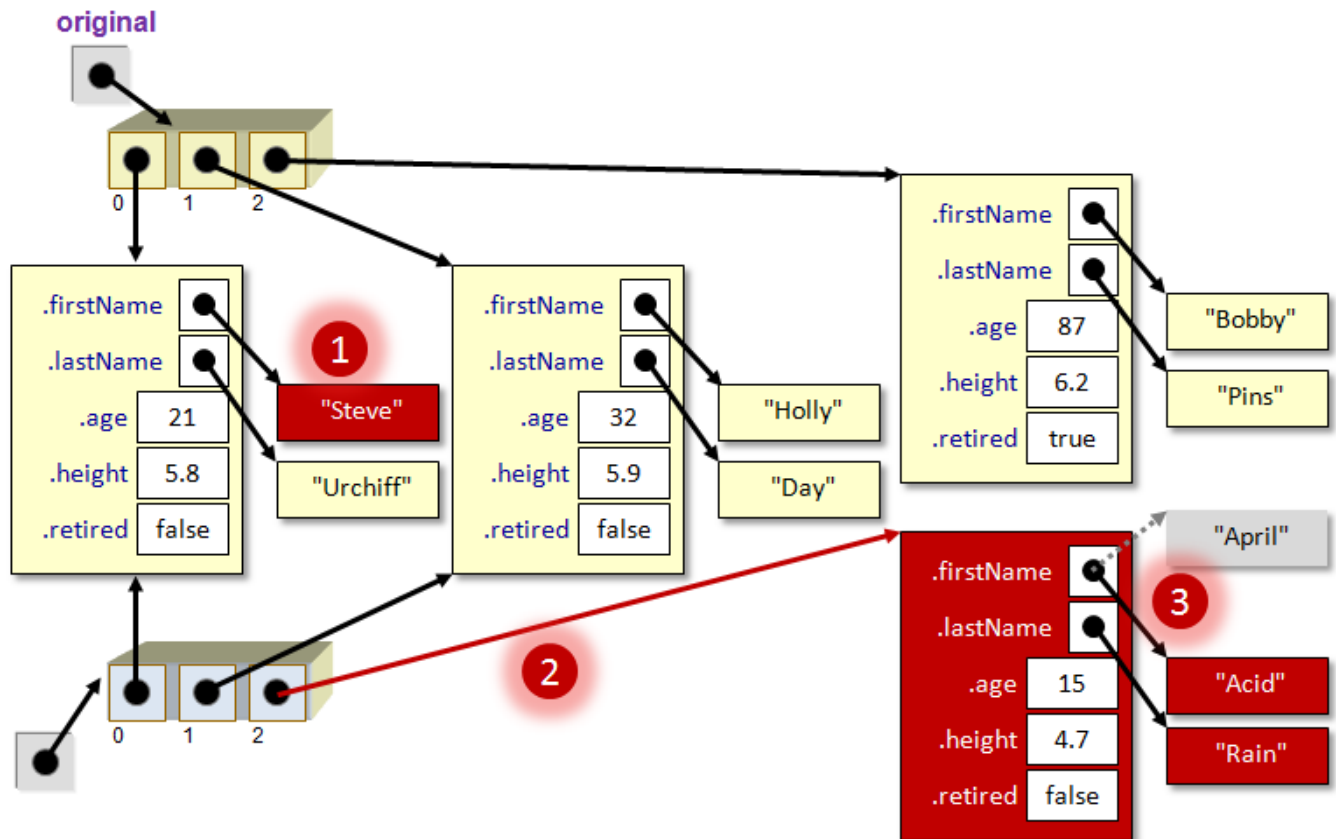
We are doing three things to the copy:

(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:

```java
public static Person[] makeCopy(Person[] aList) {
      Person[] theCopy = new Person[aList.length];

      for (int i=0; i<aList.length; i++) {
            theCopy[i] = new Person();
            theCopy[i].firstName = aList[i].firstName;
            theCopy[i].lastName = aList[i].lastName;
            theCopy[i].age = aList[i].age;
            theCopy[i].height = aList[i].height;
            theCopy[i].retiredStatus = aList[i].retiredStatus;
      }
      return theCopy;
}
```

Notice that much more work is involved.   The result for our test case would be as follows:
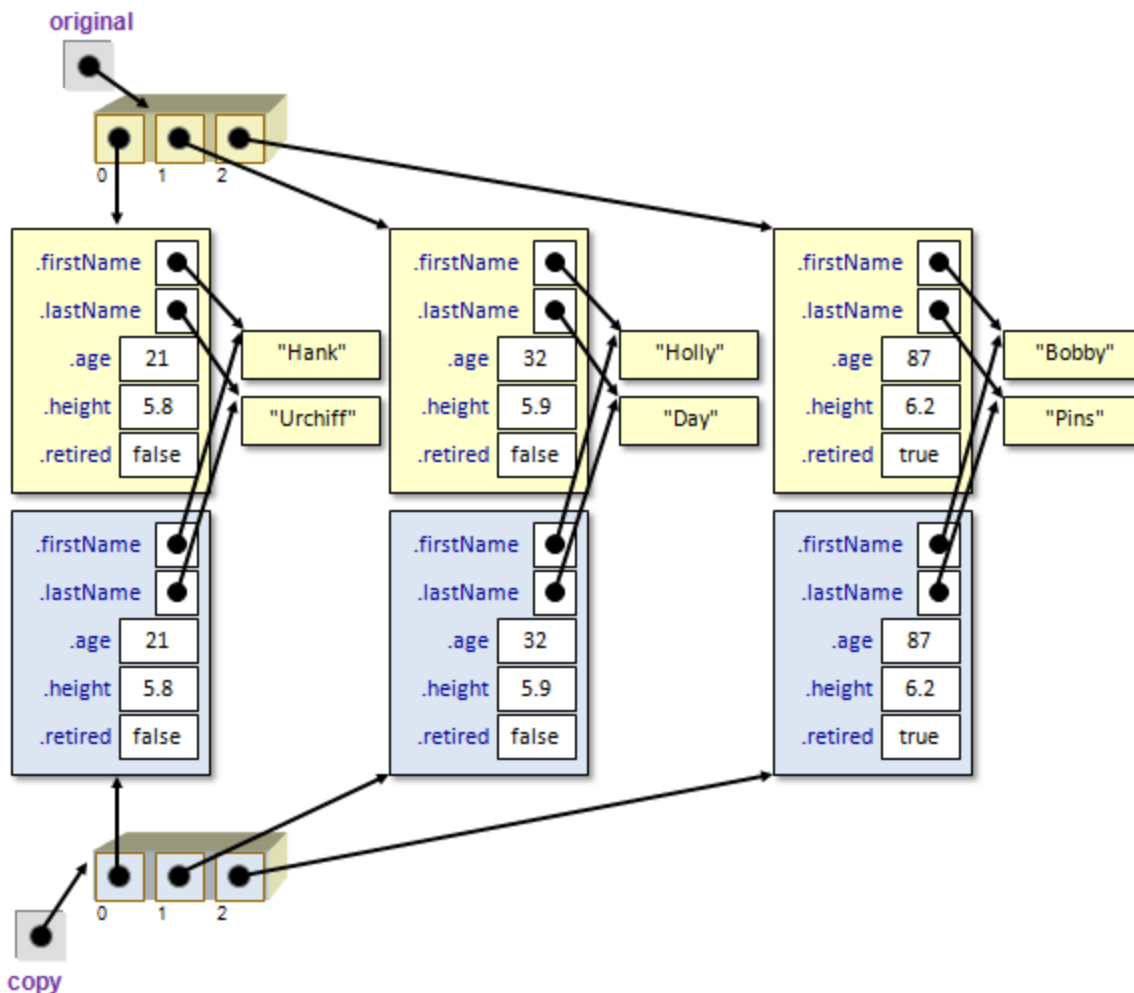
Notice that there are unique **Person** objects now and that changing the name of anyone in the copy will not affect the original.   You may notice though, that the Strings are still shared!   That means, if we altered any characters in one person's name in the copy, this would affect the original (i.e., remember...*replacing* shared objects does not cause problems but ***modifying*** shared objects does cause a problem).

Fortunately, in 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 the following lines in the **makeCopy** function:

```
theCopy[i].firstName = aList[i].firstName;
theCopy[i].lastName = aList[i].lastName;
```

into these lines:

```
theCopy[i].firstName = new String(aList[i].firstName);
theCopy[i].lastName = new String(aList[i].lastName);
```

which will make unique Strings with the same characters.   Then, we would have a true copy:
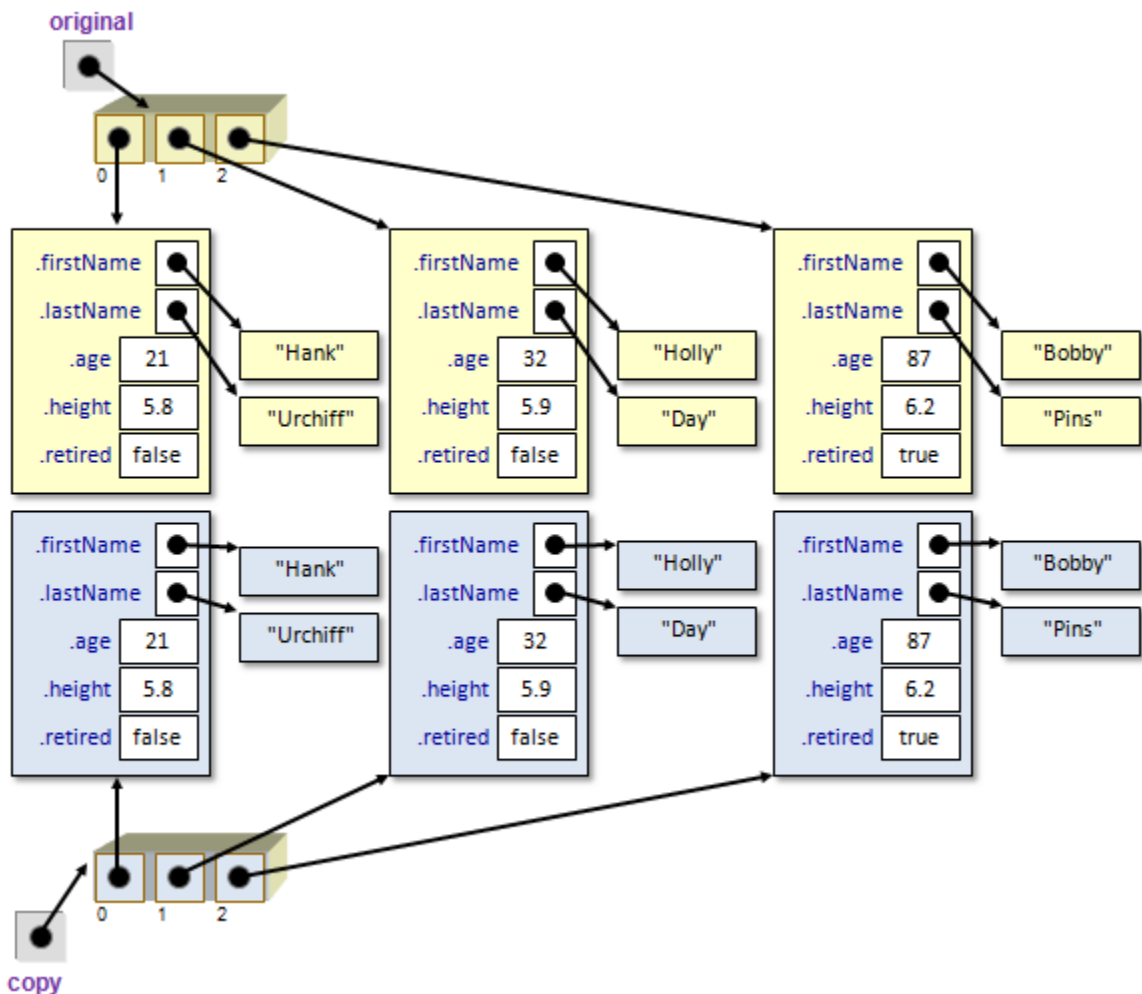
In fact, in order to make a truly deep copy, we would need to ensure that we *thoroughly* copy each of the attributes of all objects.   That is, if an object is made up of other objects ... we must go into those other objects and make deep copies of them as well.