CS1335 Documentation

Release 1.0

Karen Doore

August 25, 2015

Contents

Contents:

CHAPTER 1

Introduction

1.1 Course Overview

1.1.1 Computer Science 1 for ATEC Students

Introduction to object-oriented software analysis, design, and development. Concepts include: classes and objects, object composition and polymorphism, sorting and searching, Strings, inheritance and interfaces, and user interaction.

1.1.2 Processing

In this course we will use the Processing language. Projects will explore programming in the context of media, generative art, game design, and interaction design.

1.1.3 Textbook

The textbook for the course is Learning Processing by Daniel Shiffman.

CHAPTER 2

Variables

Variables can be considered as named containers to hold values that can be modified. Since Processing is based on the Java language, it uses *Typed Variables*. Variables can hold *primitive* values which involve a single piece of information like integers: `int`, decimal numbers: `float`, booleans: `boolean`, and characters: `char`.

2.1 Declaration and Initialization

In the example code below, the first line of code prints the sum of 2 integer literal values. In the code that follows, `int` and `float` variables are declared and assigned values.

```
println( 5 + 7 );  //two integer literal values are added together

int num1;  //declare an integer variable

int num2 = 5;   // declare an integer and assign it a literal integer value

float num3 = 5.0 //declare a floating point variable and assign it a literal decimal value;
```

2.2 Typed Variables

When using P5js and the Khan-Academy Javascript Tutorials, variables were all of the type `var`. There was no distinction between different types of variables. However, with Processing, all variables must be declared as a specific data-type such as `int`, `float`, `boolean`, `char`, etc. Typed variables allows the computer to allocate enough memory to hold the value.

2.3 Integers

Whole numbers like -1, 0, 1, 2.

2.3.1 Integer Division

When using the math division operator with integers, the resulting value is also an integer, so any fractional division remainder is truncated:

```
println( 5 / 3 );      // 1

int num1 = 5 / 2;    // 2   the remainder from division is truncated.

int num2 = 5.0 / 2  // error cannot convert from a float to an int
```

2.4 Floats

Decimal numbers like 1.0, 5.5, -1.0. Also, whole numbers like 1, 0, 1 can be stored as floating point numbers.

When initializing floating point numbers which are created using math operators, it's important to realize that integer division can cause unexpected results. Multiplying each division expression by the `float` value *1.0* can help insure no truncation occurs.

In the code below, since both 2 and 5 are written as integer literals, then expression 5/2 is evaluated using integer division. Make sure that at least 1 value is a decimal value to insure correct division of numbers assigned to `float` variables:

```
float someFraction = 5 / 2;    // 2.0   integer division of 5/2 is truncated so the result is 2.0

float correctFraction = 5.0 / 2;     // 2.5

float correctFraction2 = 5 / (2 * 1.0)  // 2.5  multiplication by 1.0 insures decimal division
```

2.5 Integer and Float Type-Conversion

Care must be taken when using `float` and `int` variables in expressions or mathematical operations together, particularly when doing division. In general, an error will be generated if an operation will result in truncation. Processing can automatically convert an `int` to a `float` value, however an error will occur when trying to convert a `float` value to an `int` value.

```
int num1 = 5;
int num2 = 2;

float val1 = num1 / num2;        //2.0  integer division expression is evaluated then assigned to val

int num3 = val1;         // error cannot convert from a float to an int
```

Processing provides type conversion functions to allow conversion between `int` and `float` variable types. There are 2 different but equivalent syntax conventions for type conversion displayed in the example code below:

```
float val1 = 5.2;

int num1 = int( val1 );  // 5   With this syntax, int( ) works like a function to convert a float va

int num2 = ( int )val1;   //5   This syntax also works to convert a float to an  int, and results in
```

2.6 Modulus Operator

The modulus operator `%` calculates the remainder of integer division. Modulus is often used to determine if a number is odd or even where n % 2 equals 0 if n is even.:

```
println( 5 % 2 );    // 1        2 goes into 5 two times with a remainder of 1
println( 5 % 3 );    // 2        3 goes into 5 one time with a remainder of 2
println( 12 % 2 );   // 0        test to determine if 12 is even, for any number n, if n % 2 = 0 then
println( 2 % 5 );    // 2        5 goes into 2 zero times with a remainder of 2
```

2.7 Booleans

Boolean variables can have the value `true` or `false`; Boolean variables are useful for storing the state some program element to control some branch option within the program, often within a conditional branch, the boolean variable value is changed to indicate the state of the program has changed.:

```
var isActive = true;
if(isActive){
        doSomething();  //trigger some state dependent behavior
        isActive=false  //change the state variable after the state behavior has been triggered
        }
```

2.8 Characters

Single letters or other unicode symbol like 'a', 'b', 'A', '%' . The `char` variable type must use single quotes around a single character. When multiple characters are used in a single variable, then the :code: *String* variable type should be used.

```
char someChar = 'a';
char otherChar = '&';
```

2.9 Random Numbers

The `random( )` function in Processing can be used to generate psudo-random variables. The `random(float min, float max)` function takes 2 input parameters and returns a floating point number ranging from the first argument to the second argument. If only 1 argument is used, then 0 is the default minimum value.:

```
float randVal1 = random( 1 , 100 );   //returns a float between 1 and 100.
float randVal2 = random( 100 ); //returns a float between 0 and 100.
```

2.10 Questions

What are the values of the following?

1. `int num1 = 2 % 10;`
2. `int num2 = 10 % 2;`
3. `int num3 = int(4.999);`

CHAPTER 3

Functions

Functions allow modular design and reusability of program components.

Functions should be designed to perform a well-defined, specific task. Functions should be designed so that they are not inter-dependent on code external to the function and so that they don't cause unintended side-effects to code outside of the function.

3.1 Function Syntax

When writing a code for function, the following components define the syntax of a function definition.

```
returnType functionName( int arg1, float arg2){  // int and float parameter arguments
        // body code of a function
}
```

For an example function that adds an `int` and a `float` values, the function syntax is

- **function name:** `addNumbers`
- **function return type:** `int //the variable type of the function's return type must be declared`
- **function arguments:** `int arg1, float arg2 //arguments must have a declared variable-type`

```
    int addNumbers( int arg1, float arg2){   //function signature
            int sum= arg1 + int(arg2);
            return sum;
    }
```

3.2 Functions and Variable Scope

In Processing, variable scope is defined by code blocks which are enclosed within curly brackets: `{ }`. When designing functions, it's important to understand that function input parameters and any variables declared within the function body are local variables to the function. When a function is executed, those variables are initialized for use within the function, but when the function execution terminates, those variables are effectively destroyed, and the memory space is returned to the computer system so it is available for use by other processes. In contrast, global variables exist for the entire life of the program execution, they aren't destroyed until the program terminates.

When designing programs and functions, it's important to consider which variables should be global, there should be a compelling reason why any variable has global scope, **most variables should be local variables**.

3.3 Function Arguments

When designing functions, it's helpful to think of the input parameters as being values that you'd like to have access to modify from outside the function. Often when designing functions, as you iterate through several design steps, you may decide to add more input parameters to your function so that you have more flexibility when calling the function. If the Processing `rect()` function only had x,y position as input parameters, then we'd be quite limited in how we could use the function. The addition of width and height parameters gives more flexibility. There's also another version of the `rect()` function that takes and additional parameter to specify the radius of corners so you can create rounded rectangles, this is an example of function overloading which is explained below.

```
rect(float x, float y, float width, float height, float radius); // rounded rectangle version of the
```

3.4 Function Overloading

Often when designing functions, we can design multiple versions of a function, where each version of the function takes a different number or type of input parameters. In Processing this conveniently allows for several different versions of the fill function.

```
fill(float grayScale);   //one input parameter corresponds to grayscale colors

fill(float redVal, float greenVal, float blueVal);  //three input parameters corresponds to RGB.

/* Using a global state variable, set with colorMode(), allows the same fill( ) function signature
create HSB colors while still using the same function definition. */

colorMode(HSB);   //the the colorMode function changes a global color-state variable to control how

fill(float hueVal, float saturationVal, float brightnessVal);  // now the fill color is HSB
```

3.5 HSB Color-Slider Example

The *HSB Color-Slider Example* project creates an interactive HSB color selector to demonstrate the iterative design process for designing functions. In order to create this UI-widget, first we need to figure out the required components. First, let's plan to create a simple rectangle that is filled with the full HSB hue-range. Then we'll need to figure out how to let the user interact with it to select a color. Then we'll want provide a way to use that selected color in another part of the program.

- **Input values:** position and size of the slider widget.
- **Output values:** a hueValue that has been selected.
- **Display:** some representation of a range of hues, and indication of currently selected hue value.
- **Interactivity:** a means for the user to modify and select a hue value.

Project and Code: *HSB Color-Slider Example*

The image below shows the widget we'll design in this code project.

CHAPTER 4

HSB Color-Slider Example

4.1 Overview

The project below creates an interactive HSB color selector as an example project to demonstrate the iterative design process for designing functions. In order to create this UI-widget, first we need to figure out the required components. First, let's plan to create a simple rectangle that is filled with the full HSB hue-range. Then we'll need to figure out how to let the user interact with it to select a color. Then we'll want provide a way to use that selected color in another part of the program.

- **Input values:** position and size of the slider widget.
- **Output values:** a hueValue that has been selected.
- **Display:** some representation of a range of hues, and indication of currently selected hue value.
- **Interactivity:** a means for the user to modify and select a hue value.

The image below shows the widget we'll design.

4.2 HSB ColorMode

HSB Color works well for a color picker because it allows for a full color spectrum to be represented by making incremental changes to a single color variable. In Processing, we can use the `ColorMode(HSB)` function so that the `fill()` and `stroke()` functions can use the HSB color values instead of the default RGB color values. Whereas when using the `fill(float redVal,float greenVal,float blueVal )`, the input values specify red, green, and blue color components. With HSB, the 3 input parameters for `fill(float hueVal,float saturationVal, float brightnessVal)` specify hue, saturation and brightness values.

For the HSB slider, we'll keep the saturation and brightness fixed at the maximum values of 255, so the only value that changes for this widget is the hue value, which will range from 0-255. The image below shows that the top surface of the HSB color-space cone represents values where the Brightness value is maximum. The top surface of the cone represents a color wheel with saturation varying from 0 at the middle of the cone to a maximum saturation at the outer perimeter of the wheel. At the outer perimeter, the saturation and brightness are at the maximum values, the hue varies in values from 0 to 360 degrees. So, the minimum value of hue = the maximum value of hue = Red.

Our color slider (above) also shows that red occurs at the both the maximum and minimum values. Although the circular values for hue range from 0 to 360 degrees, we will use the Processing convention of having color values that range from 0 - 255, which corresponds to 8 bits of color information for each color input parameter. We will have to insure that our slider shows the full range of hue values, so we'll need to transform the 0-255 values to fit within the width of our rectangular slider `sWidth` dimensions.

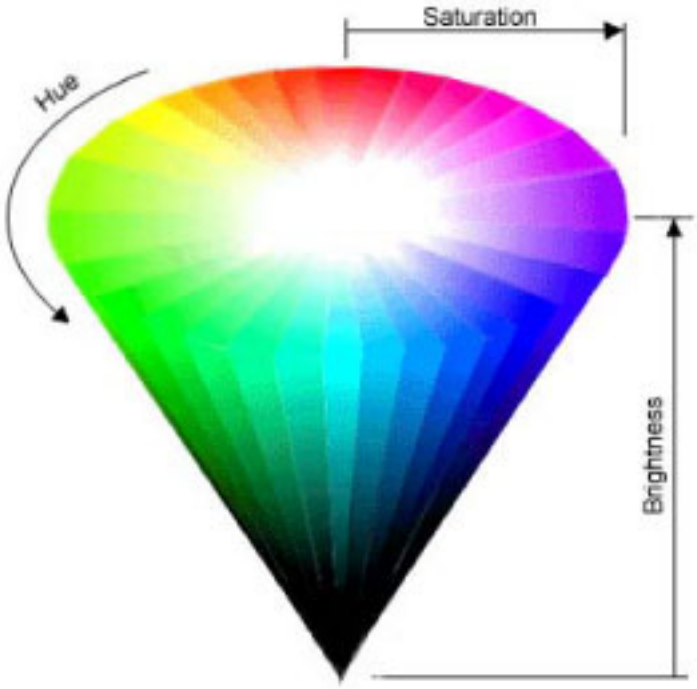


Fig. 4.1: Image from: TomJewett.com

4.3 Function Declaration

Because this will be an interactive widget, then we'll need to use the processing `draw()` loop. We'll also start with the `setup()` function so we can set the canvas size. We also know that we're going to create a function that displays an HSB Slider so we can create the declaration that function using the information we identified above:

```
void setup(){
size(300,100);
}

void draw(){
        float hueValue = drawSlider(20,20,200,50);  //draw the slider at x=20,y=20, 200px wide, 50 p
}

//function to create interactive HSB slider, returns the selected hueValue
float drawSlider(int xPos, int yPos, int sWidth, int sHeight){

}
```

4.4 Function Design - Step 1

So, now we need to think about what code we will write inside of the function. The things we need to do in the function are:

- Draw an HSB hue gradient inside a rectangle to show available colors for selection.
- Draw another small rectangle to show the selected color.
- Draw an indicator rectangle that slides along the slider to modify the color.
- Write code to allow changing values when the user drag the slider indicator.

4.5 Hue Spectrum- Rectangle

The first thing to do is to draw a basic rectangle using our function's input values. Then we need to figure out how to create a RoyGBiv rainbow spectrum inside the rectangle that corresponds to the full range of hue values from 0-255. One idea is that we could fill the rectangle with points of color, where we vary the hue value of each point along the x and y values of the rectangle. However, for our needs, we can actually use lines since we don't need to vary the color along the y-axis, all points in the rectangle that have the same y value will have the same hue value. So, to color lines, we need to use the `stroke()` function in Processing. We'll want to use a loop, and for each value of x in the rectangle, we'll want to vary the hue value. We can do that with a for loop, where each value of i corresponds to a 1 pixel increment in the x direction. If our rectangle was 255 pixels wide, each pixel would represent 1 possible hue value. That code would be something like this:

```
colorMode(HSB);   //set the colors to HSB

for( int i = 0; i <= 255 ; i ++ ){

        stroke( i , 255, 255 )  // i is hue value, 255 is max value for saturation and brightness

        line(i, yPos, i, yPos + sHeight )  // the line is vertical at x=i, y values are yPos, and yPo

}
```

This would work fine if we always wanted to have our slider have a width of 255 pixels, however we'd like to give ourselves more flexibility so that we can create sliders of any width, based on the input value sWidth of our function parameters. We basically need to determine the fractional position for each location and multiply that by the max hue value of 255. We could write a separate function to do that calculation like below

```
float hueMapping( int i, float sWidth ){  // i is the current value of 'i' in the for loop

        return ( i / sWidth ) *  255 ;  // will return values in range of 0.0 - 255.0
}
```

Then we could use in the following manner in the for-loop of our function

```
for( int i = 0; i <= 255 ; i ++ ){

        float hueValue = hueMapping( i, sWidth) // local variable within for loop: values in the rang

        stroke( hueValue , 255, 255 )  // i is hue value, 255 is max value for saturation and brightn

        line(i, yPos, i, yPos + sHeight )  // the line is vertical at x=i, y values are yPos, and yPo

}
```

4.6 Map() Function

This type of calculation is a mapping between 2 value ranges, we have a *current* range of 0-sWidth of the rectangle and we need to map that to the *target* range of hue values which is 0-255. This is such a common type of calculation that Processing provides us with a function to do this called: `map()` with the function signature: `map(value, start1, stop1, start2, stop2)` The map function takes 5 values: the first parameter is the actual value you're trying to determine the mapping for and the other 4 parameters are the min-max values for the 2 different numeric ranges which are the *current* and *target* ranges; the return value is the answer for your conversion calculation, so in our case we'd use:

```
float hueValue = map( i, 0.0 , sWidth , 0.0 , 255.0 );  //current range is the 0-sWidth, target range
```

So we can actually just use the map function, no need to create our own function mapping function. The benefit of using a separate function for such a calculation is that we can test code to make sure it's working correctly.

4.6.1 Inline Calculation with Integer Values

If we had simply done the conversion as an inline mathematical operation, it might be difficult to track down any math errors such as truncation issues since we had declared our sWidth to be an int variable type

```
float hueValue= ( i / sWidth ) * 255;  // hueValue=0.0  division with integers i and sWidth causes ti
```

Therefore, let's change our function signature so that the input variables are all floats, just to insure any division operations using these values results in a correct value. So, now we should modify our drawSlider function as below:

```
float drawSlider( float xPos, float yPos, float sWidth, float sHeight )  //new function signature usi
```

4.7 Interactivity

We have decided to provide a narrow rectangle to represent the interactive component of the slider, so we need to create this rectangle and set it's fill color to the currently selected hueValue. Then we'll need to use a conditional statement to determine when the user is moving the slider so that we can change the hueValue. The code below tests to see if a users mouse is within the rainbow filled rectangle and if the mouse is pressed, if this is true, then we need to store the x-location of the mouse within the rainbow filled rectangle. We will use the variable sliderPos to store the position of the pressed-mouse, and we need to subtract the rectangle's xPos so that we're recording the location within the rectangle, not the mouseX position relative to the overall canvas as seen in the image below this code example:

```
if(mousePressed && mouseX>xPos && mouseX<(xPos+sWidth) && mouseY>yPos && mouseY <yPos+sHeight){
        sliderPos=mouseX-xPos;  //only change sliderPos if the user is within the slider area
                hueVal=map(sliderPos,0.0,sWidth,0.0,255.0);  // get new hueVal based on moved slider
}
```

4.8 Global Variables

The `sliderPos` variable is used to capture the current location of the slider due to the user's interactivity, but we need to keep track of this value between each function call, so we know where to position the slider, and what the current selected hueValue is. We also need a means to initialize this value before it's been moved by the user. These 2 variables are related to each other as discussed above when using the `map()` function. We have determined that we need access to the hueValue in the draw loop so we can use it to set color for other items that the user draws, but we really don't have a need for this `sliderPos` value outside of this function, therefore, it should be a local value to this function, it has no meaning outside of this function.

However, the _hueValue variable should be a global variable because it will be an input to our function as well as an output value from our function, and we need to set an initial value for the slider so that there's a default color even if a user hasn't touched the slider. However, if we created and initialized it as a local variable within the draw loop then it would be re-initialized each time the draw loop code was executed, therefore, this is justification for why we can declare this as a global variable, although there would be other ways we could achieve this initialization and we'll cover that when we learn object-oriented program design. We'll name the global variable using an underscore at the beginning of the word `_hueValue` so that we can distinguish it from the local `hueValue` variable that we're modifying within the drawSlider function itself. The `sliderPos` value will be created and initialized using the map() function each time our function executes, based on the input value of `hueValue`, it will only change if the user drags the slider.

So, we need to modify our function so that we're passing hueValue in as a parameter, and then we return that value from the function so that any changes to the hueValue caused by user-interaction with the slider are reflected in the updated global _hueValue. Below is our final function signature, and the complete code is shown at the bottom of this page:

```
float drawSlider(float xPos, float yPos, float sWidth, float sHeight,float hueVal){
        //function body
        return hueVal;
}
```

Finally, the last bit of code for this slider is that we want to draw a white rectangle behind our slider, so our animation doesn't have 'trails'. We don't want to use a background(255) in the actual draw loop because we want to allow the user to be creating drawings when dragging the mouse. Below is the final code for this slider. We also have put a `fill()` and `rect()` functions in the draw loop to verify that shapes drawn in our app are being updated as the slider is moved.

4.9 Final Version of Code

```
//global values

float _barWidth=300.0;    //slider-bar width;
float _hueVal=_barWidth/2;  //initial hueValue global value

void setup(){
  background(255);
  size(400,400);
  colorMode(HSB);
  stroke(0,0,0);
}

void draw(){
  _hueVal= drawSlider(20.0,300.0,_barWidth,30.0,_hueVal);
 fill(_hueVal,255,255);  //use the new _hueValue to set fill for drawing canvas object
 rect(10,10,100,50);  // verify color changes when slider is moved
}

float drawSlider(float xPos, float yPos, float sWidth, float sHeight,float hueVal){
  fill(255);
  noStroke();
  rect(xPos-5,yPos-10,sWidth+10,sHeight+20);  //draw white background behind slider

  float sliderPos=map(hueVal,0.0,255.0,0.0,sWidth); //find the current sliderPosition from hueVal

  for(int i=0;i<sWidth;i++){  //draw 1 line for each hueValue from 0-255
      float hueValue=map(i,0.0,sWidth,0.0,255.0);  //get hueVal for each i position //local variable
```

```
      stroke(hueValue,255,255);
      line(xPos+i,yPos,xPos+i,yPos+sHeight);
  }
  if(mousePressed && mouseX>xPos && mouseX<(xPos+sWidth) && mouseY>yPos && mouseY <yPos+sHeight){
     sliderPos=mouseX-xPos;
     hueVal=map(sliderPos,0.0,sWidth,0.0,255.0);  // get new hueVal based on moved slider
  }
  stroke(100);
  fill(hueVal,255,255);  //either new or old hueVal
  rect(sliderPos+xPos-3,yPos-5,6,sHeight+10);  //this is our slider indicator that moves
  rect(sWidth+40, yPos, sHeight,sHeight); // this rectangle displays the changing color to the right
  return hueVal;
}
```

4.10 Questions

1. Can you create a Saturation Slider to let the user change the HSB saturation value.
2. Can you create a Brightness Slider?
3. Can you create an Alpha Slider?
4. What representations can you use so the user understands how interaction with these sliders changes their selected color?
5. Can you create a small random variation in these values so when the user draws artwork, the colors show very slight random variation?
6. How can you incorporate these sliders into a drawing program so the user can create interesting artwork?
7. Why is it better to have hueValue as an input parameter to our function rather than modifying the global variable _hueValue within the function itself?
8. How would you use the `map()` function to determine the xPosition for the indicator rectangle if you only know the current hueValue?
9. When using i as the x-postion of our colored lines that fill the slider rectangle, what adjustments do we need to make to insure that we can draw our slider anywhere on the canvas. How can i be determined, relative to the slider xPostion?

CHAPTER 5

Button Behaviors

In this project, we'll look at button states like hover, pressed, and clicked, as well as how buttons can be used to provide users with control over states of other elements of our programs. This project continues to emphasize the use of functions to create modular code. In a future version of this project, we'll create object-oriented buttons. As this project shows, it is helpful to think of buttons as having different behaviors such as display color depending on their current state and user interaction. It is instructive to consider how a button differs from a simple rectangle, code written for the button directly relates to the behaviors and functions of a button when contrasted with a simple rectangle.

Below is one possible way we could describe or model the features of a button:

Function:

- Indicate and allow change of *State* for some system variable

Behaviors:

- Default, Hover, MousePressed, MouseClicked

Structure:

- Square defines activation area
- Circle fill and stroke changes to indicate behavior and state
- Position is defined with x,y coordinates on the canvas
- Size is defined by some width and height values

Here, we will use global variables to maintain these button states, yet intuitively, it would make sense that the button display state information should only be used within the code to display the button.

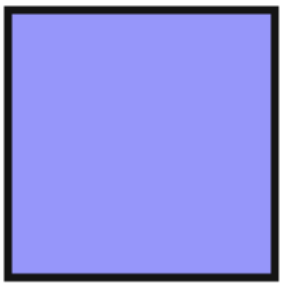
In the code below, we create a very simple program with 2 rectangles to explore how to design a button. We want the button to respond to the user's mouse when the mouse interacts with the button's rectangular area to give the user indication that the button is interactive. We will have 2 types of dynamic behavior in this project, one set of behaviors relate to how the button rectangle and circle change with mouse interaction. Then, we need to have the button change the state of the blue rectangle, using the global state variable `rectState`

```
int rectState=1;  //state variable to control fill color of rectangle

void setup(){
        size(250,150);
}

void draw(){
        background(255);
        switch(rectState){
        case 0:
```

```
                fill(0,0,255);  //blue
                break;
        case 1:
                fill(150, 150,250);  //light blue
                break;
                default:
                        fill(100);  // if neither state, have a gray rectangle
                        break;
        }
        rect(10,10,100,100);  // draw rectangle which will be controlled by button
        drawButton(150,40,50,50);
}

void drawButton(float xPos, float yPos, float bWidth, float bHeight){
        fill(100);
        stroke(20);
        strokeWeight(3);
        rect(xPos,yPos,bWidth,bHeight);
        fill(255,0,0);
        ellipse(xPos+25,yPos+25,bWidth-5,bHeight-5);
}
```

The `switch` statement provides a variation on the conditional branching structure, we could use `if()`, `else`, but for some situations, `switch` provides a cleaner, simpler structure.

Our code currently creates this program display where the button will control changing the color of the light blue rectangle:



5.1 MouseOver

Now we need to create code that will respond to a user's mouse being over the rectangle, so let's create a function to test if the mouse is over the button. The function input parameters will be the button's position and shape, we want to have it return a `boolean` `true` or `false` value

```
boolean isMouseOver(float xPos, float yPos, float bWidth, float bHeight){
     if(mouseX> xPos && mouseX < xPos + bWidth && mouseY > yPos && mouseY < yPos+bHeight){
                        return true;
        }
        return false;
}
```

5.2 Responsive Button

Now we need to use our `isMouseOver` function to add some interaction feedback to the user. We will consider 4 different states of our button where we will design our button to have toggle behavior, this means that it changes state from active to default every time the user clicks on it:

- **default: button is inactive**
  - red `fill` circle and black `stroke` outline

- **hover: the mouse is over the button, but not pressed or clicked**
  - red `fill` circle and white `stroke` outline

- **pressed: the mouse is over the button and is being pressed** -dull green `fill` circle and white `stroke` outline

- **clicked: the user clicked on the button to put it in the active state.** -bright green `fill` circle and white `stroke` outline

5.3 Mouse Event Handlers

We can use nested `if` blocks and the `isMouseOver` function to add this logic to the `drawButton` function. First, we'll want to define a variable that we can use to track the buttonState as `_btnActive`. Since this value will be used in the draw loop and needs to have be initialized in the `setup()` function, we will make it a global variable. Similarly logic can be used to justify creating a global varialbe `_btnHover` that will be used to track whether the user's mouse is over the button. We can use this variable in the `mouseClicked` function to determine if the button state should be modified by the mouse-click. The following code is part of the drawButton() function and controls the coloring of the button circle as identified above.

```
// inside drawButton() function

_btnHover=isMouseOver(xPos,yPos,bWidth,bHeight);
```

```
if(_btnHover){
        stroke(255);    //white outline
        if(mousePressed){
                fill(160,200,0);        //dull green
        }
}

if(_btnActive){
        fill(100,200,0);        //green
        stroke(255);
}

ellipse(xPos+25,yPos+25,bWidth-5,bHeight-5);  //draw the button's circle with fill that was executed
```

5.4 MouseClicked

We also need to create the code to toggle the state of `_btnActive` when it the mouse is clicked and the mouse is located directly over the button. We can use the global variable `_btnHover` as an initial conditional check to determine if any action needs to be executed, otherwise, the user has clicked outside of the button area.:

```
void mouseClicked(){
  if(_btnHover){  //only change btnState if the user is over the button when clicking
      if(_btnActive){
          _btnActive=false;
          rectState=1;
      }
      else {
          _btnActive=true;
          rectState=0;
      }
  }
}
```

Here is the processing sketch:

Below is the full code to create a responsive button that controls the behavior of a separate rectangle.:

```
int rectState=1;
  boolean _btnActive=false;
  boolean _btnHover=false;

  void setup(){
    size(250,150);
  }

  void draw(){
      background(255);
      switch(rectState){
        case 0:
           fill(0,0,255);  //bright blue when button is active
           break;
        case 1:
          fill(150, 150,250);  //light blue when button is off
          break;
      default:
        fill(100);  // if neither state, have a gray rectangle
```

```
        break;
    }
    stroke(20);
    rect(10,10,100,100);  // draw rectangle which will be controlled by button
    drawButton(150,40,50,50);
  }


void drawButton(float xPos, float yPos, float bWidth, float bHeight){
      fill(100);
      stroke(20);
      strokeWeight(3);
      rect(xPos,yPos,bWidth,bHeight);
      fill(255,0,0);

      _btnHover=isMouseOver(xPos,yPos,bWidth,bHeight);
      if(_btnHover){
        stroke(255);  //white outline
        if(mousePressed){
          fill(160,200,0); //dull green
        }
       }
       if(_btnActive){
          fill(100,200,0);  //green
          stroke(255);
       }
      ellipse(xPos+25,yPos+25,bWidth-5,bHeight-5);
 }

boolean isMouseOver(float xPos, float yPos, float bWidth, float bHeight){
       if(mouseX> xPos && mouseX < xPos + bWidth && mouseY > yPos && mouseY < yPos+bHeight){
        return true;
    }
    return false;
}

void mouseClicked(){
  if(_btnHover){  //only change btnState if the user is over the button when clicking
      if(_btnActive){
          _btnActive=false;
          rectState=1;
      }
      else {
          _btnActive=true;
          rectState=0;
      }
  }
}
```

CHAPTER 6

Drawing Application

This project builds on the previous 2 projects where we implemented code to create a hue color slider and an interactive button. Both of these previous projects emphasize the use of functions to simplify programs. In this section we'll explore some options for a creative drawing application. As with many drawing programs, when the user drags the mouse, we'll write a program to create a brush-type of pattern. We'll use functions to provide structure and organization for our code.

If we imagine ourselves as the brush in a drawing application, we should consider what information we'd have available as input, each time there's a new frame in the drawing loop. We'll have access to the global values: `mouseX` and `mouseY` and that will determine the location of the shapes drawn in a current time-frame. We also have access to the global values: `pmouseX` and `pmouseY`, which represent the location of the mouse in the previous execution time-frame of the draw loop. Using these values together can allow us to create more interactive drawing patterns.

6.1 Mouse Speed

In Learning Processing chapter 3, exercise 3.6, Daniel Shiffman uses the mouseX, mouseY and pmouseX, pmouseY variables to draw a line following the mouse movement:

```
line( pmouseX, pmouseY, mouseX, mouseY);
```

We can determine the distance that the has mouse moved since the last frame by observing that in the x direction, the mouse has moved the *absolute value* of (mouseX-pmouseX) and the same can be determined in the y direction. Absolute value gives us the positive difference between 2 points. Since `(pmouseX-mouseX)` might be a negative value, depending on which direction the mouse was moving, but we're just interested in the magnitude or amount of movement, then we need to use the absolute value function. This provides a few interaction parameters that we can use to create a more interactive drawing brush than just drawing a line between successive mouse positions. So, the speed of the mouse would be the distance traveled in a given amount of time. We can use the fact that the time between frame execution is a measure of time, so one measure of speed would be:

```
float speed = abs(pmouseX-mouseX) + abs(pmouseY -mouseY); //abs is absolute value or magnitude of di
```

Then we can use that speed value to control some aspect of the elements drawn. In Shiffman's example, he suggests using speed to vary the value of the `strokeWeight`, below is one possible expression which could create an interesting drawing brush.:

```
strokeWeight( 1 + (.05* speed));
```

6.2 Mouse Distance

Processing provides a distance function we can use to determine the distance between points. It takes as input, the x,y positions of 2 points. We can use the `pmouse` and `mouse` positions to determine the distance between 2 points, or we can create some global position variables `_x`, `_y` and use those to determine distance from the mouse position. This will allow us to control how far the mouse must move between drawing positions. If we were to draw ellipses at the current mouse position, and only want to allow the drawing application to allow drawing another circle if the mouse has moved atleast some minimum distance between each circle before drawing the next circle, these global location values can provide more control than using `pmouseX`, `pmouseY` which are updated by the system each frame execution.

```
float _x;  //global variables which are initialized outside of draw
float _y;  // _x, _y mark the location of the last drawn element, only updated when an element is dra

  void setup(){
        size(400,400);
        _x=width/2;
        _y=height/2;
                background(255);
  }
  void draw(){
        float distance=dist(_x,_y,mouseX, mouseY);
        if((distance > 10) && mousePressed){  //make sure  mouse has moved 10 pixels between the sta
                ellipse(mouseX,mouseY,20,20);
                        _x=mouseX;   //update value of last draw position
                        _y=mouseY;
        }
  }
```

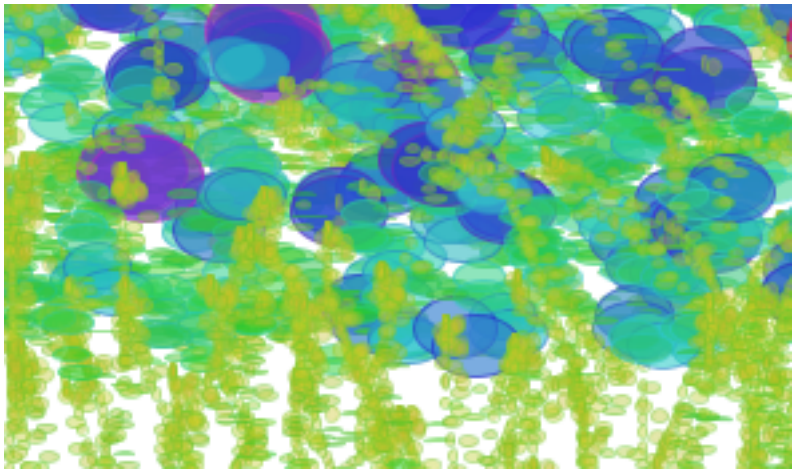
6.3 Creative Brushes

One thing to consider is the range of possible values when using a parameter like speed to create variation in a drawing feature. If no mouse motion occurs, then speed=0. Using `println(speed)` is a good way to see the range of values for typical mouse motion. Once that's been determined then it's easier to find an interesting way to integrate speed into the drawing program. We can use speed to modify values of color, alpha, shape dimensions, scale, rotation angle etc. If we use any transformations in our drawing app, then we'll want to use `pushMatrix()`, and `popMatrix()` to help insure that other drawn elements like our sliders which are *drawn* at the end of the draw loop, aren't distorted by any transformation we'd apply for our brush effects. Here's an example of using speed along with some small random variations to modify a wide range of values, so that just drawing 2 ellipses creates a somewhat interesting brush. One important goal is that we want the drawing brush to somewhat intuitive so that a user can realize, if I draw slower, then I get smaller shapes, this is essential in order for the brush to be useful for creating an interesting artwork:

```
void setup(){
        size(400,400);
        background(255);
        colorMode(HSB);
}
void draw(){
        float speed=abs(mouseX-pmouseX) +abs(mouseY-pmouseY);
        if(mousePressed){  //only draw if the mouse is pressed
                pushMatrix();
                translate(mouseX, mouseY);
                fill(40 +(speed*.95), 200+(random(-10,10)),200+(random(-10,10)),100 +(speed*.55));
                stroke(60 +(speed*.95), 200+(random(-10,10)),200+(random(-10,10)),100 +(speed*.55));
```

```
                ellipse(0,0,2+(speed*.25),6-(speed*.25)); //use speed to modify width, height
                ellipse(3+random(-5,5),5+random(-5,5),5+(speed*.25),6-(speed*.25)); //use speed to mo
                popMatrix();
        }
}
```

Below is a screen-shot from the brush created above where there's not even a color slider option for the user to modify. These images show that there were predictable behaviors of the brush that allowed the user to create a composition based on understanding the brush behavior, in this case: drawing pattern varied with mouse speed.



6.4 Questions

1. How can we determine some measure of the mouse speed, given the current mouse positions and the previous mouse positions?

CHAPTER 7

Buttons As Objects

In the previous section, we discussed buttons in terms of features like: structure, function, and behaviors. Then we wrote code to implement these features, and this code is what turned a simple geometric shape into a user-interface element. How is a button different than a rectangle? A button is different than a rectangle only when we write the code to make it different. It is not difficult to imagine our program's buttons as objects because we all have experience using physical buttons in our daily routines, and we have an intuitive understanding of how buttons operate.

7.1 Physical Objects

When we use a physical button, like the on-off button of a light-switch, we understand that the button has a physical configuration that controls electrical current to the bulb. The physical movement of the switch itself is responsible for controlling the state of the lightbulb as well as controlling the state of the switch itself, so we can look at the switch and understand if it's in the on or off position. However, when we think about a virtual button on a device such as our mobile phone, there is not a tangible, physical configuration of the button that controls these behaviors. To be intuitive for a user, the virtual button should behave analogously to the physical version, so it is helpful for us to think of buttons that we create as if they were physical objects.

7.2 Object Responsibilities

If we think of our button as a physical object, then it makes sense for our button object to be responsible for it's own behaviors and states. A physical button has a configuration that controls and indicates if it's on or off. For our button, we need to create a *state variable*, `on`, so the button object can remember whether it is currently on or off. Similarly, we want the button to be responsible for responding to user-events whether it is and it's display (behavior), and for knowing it's current state, size, position and orientation (state and configuration). Then we can have many button instances, each responsible for is own object properties, and the object behaviors (methods) will be consistent across all instances.

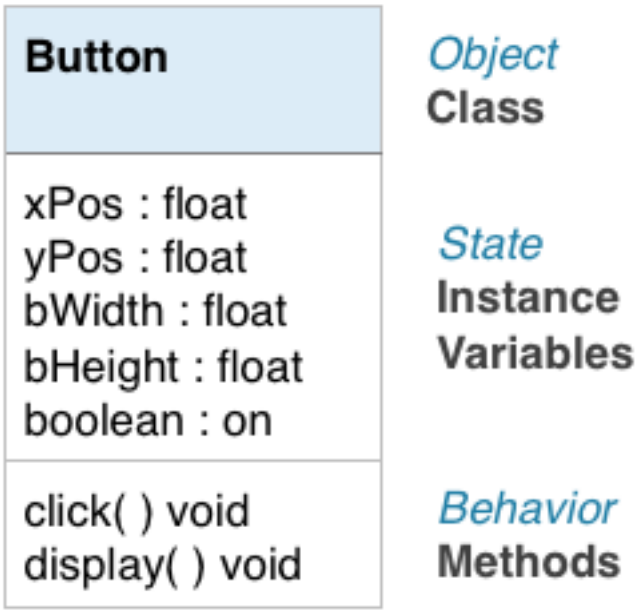
7.3 Objects and Classes

To create objects, we write code to define a **class**, which we can think of as the **blueprint** for creating objects. When creating objects in code, it's helpful to think of object responsibilities in terms of what information an object knows about itself, and the behaviors an object can do. In our code, we will use **instance variables** to store an object's state and configuration information. To implement an object's behaviors, we'll write functions, these are referred to as the object's **methods**. Once we write code to define the Button class, then we can create an unlimited number of button

objects. When our code is executed, then the button objects are created where the compiler uses the class code as the blueprint.

7.4 Object - Class Structure

The image below is a **UML Class Diagram**. UML *Unified Modeling Language* is modeling language specification that provides formal structures for designing models of systems. The UML website states that 'modeling is the designing of software applications before coding.'



The class diagram shows the name of the class, the instance variables, and the methods. In UML, we can use class diagrams to show relationships between several different classes. There are a wide variety of UML diagrams, some are designed to show structure like this class diagram, while other UML diagrams are designed to model behavior and interaction of system entities.

7.5 Processing Tabs

Processing provides tabs to allow us to organize our code when using classes, the main tab is the name of the sketch, while each other tab should be the name of the class, such as *Button*. The image below shows the Button tab with the basic code elements which define the class.

ButtonTest | Button

```
class Button{
   //class instance variables
  float xPos, yPos,bWidth,bHeight;
  boolean on;

  Button(){  //default constructor
    xPos=width/2;
    yPos=height/2;
    bWidth=100;
    bHeight=100;
    on=false; //start with button off
  }

  //class methods
  void display(){
    fill(255,0,0);
    rect(xPos,yPos,bWidth,bHeight);
  }

  void click(int mx, int my){
    //respond to mouseclick
  }

}  //end of class
```

When looking at the code in a Class definition, we can see a similarity with the structure of code that we've been writing in our previous examples. The table below shows these similarities, in the left column, we can see that the code we write in the main tab can be though of as having 3 sections, the top of our programs is where we declare global variables, then the setup() function is executed once, while the draw() function is where the main behavior of our program is typically executed. When we write code for a class definition, we are required to write our instance variables at the top, then we must write the constructor functions, these are similar to setup() in that the constructor for an object is a function that is only executed once, when the object is first created and it's used to initialize all of the instance variables for our object. Finally, the bottom of the tab contains all of the functions / methods that we write for our object's behaviors. Within these methods, we can also have local variables, but they will only exist for the duration of that method's execution. The instance variables exist for the life of the object instance, store all of the information about the state and configuration of our object instances throughout the duration of an object's lifetime.

Main Program Tab	Class Definition Tab
Global Program Variables	**Class** Instance Variables
Setup( ) *called once*	Constructor() *called once*
Draw( ) *main program* *behavior* *supplemental local variables*	Methods( ) *object* *behaviors* *supplemental local variables*

7.6 Questions

1. What is the difference between a class and an object?

CHAPTER 8

PVector

On the processing website, there is a comprehensive PVector tutorial about the PVector object. The PVector object is used to represent vector objects, and provides data elements and methods for vector math operations. We can consider a vector to be a collection of values that describe some concept that has some spatial dimension, or some properties that have directional components. A vector in mathematics represents the difference between 2 points in space. When a vector is used to represent a single point's location, it is assumed that the 0,0 origin is one of the vector endpoints. The simple ball bouncing animation below gives an example of using the ball's x,y location and adding horizontal speed and vertical speed to the balls current location to animate the motion of the ball. Location, velocity (speed), and acceleration can all be represented using the PVector object, in addition, a PVector could also represent something like friction which might have different values along different x, y, or z dimensions. The PVector object has methods to `add()` 2 vectors which simplifies working with motion concepts in multiple dimensions. In addition, the PVector object provides a good introduction to object oriented programming.

At its core, a PVector is just a convenient way to store two values *Daniel Shiffman* PVector tutorial

8.1 Bouncing Ball - No Vectors

In the PVector tutorial and in Learning Processing Section 5.7, the example programs create a bouncing ball. The bouncing ball programs provide a good example of how to program object motion. The code below is from the Processing.org *PVector tutorial* and it creates a simple bouncing ball.:

```
float x = 100;
float y = 100;
float xspeed = 1;
float yspeed = 3.3;

void setup() {
  size(200,200);
}

void draw() {
  fill(255,10);
  rect(0,0,width,height);  //transparent overlay background

  // Add the current speed to the location.
  x = x + xspeed;
  y = y + yspeed;

  // Check for bouncing
  if ((x > width) || (x < 0)) {
    xspeed = xspeed * -1;  //reverse speed
```

```
  }
  if ((y > height) || (y < 0)) {
    yspeed = yspeed * -1;   //reverse speed
  }

  // Display at x,y location
  fill(175);
  ellipse(x,y,16,16);
}
```

8.2 Bouncing Ball with PVector

The PVector object provides variables to hold the data of the ball's location coordinates and also provides methods that allow for easy use of vector operations for movement. In the code above, the ball's location is updated during each loop, so the new location = location + speed, and this is calculated separately for each of our canvas's 2 dimensions: x and y. We can instead use a PVector to hold the location and another PVector to represent speed, and then we can use the PVector add() method to calculate the new location of the ball. Below is the same bouncing ball program using PVectors for location and velocity.:

```
PVector speed = new PVector(1 ,  3.3);  // create a new PVector object instance
PVector location = new PVector(100,100);

void setup() {
    size(200,200);
}

void draw() {
    fill(255,10);
    rect(0,0,width,height);  //transparent overlay background

    // Add the current speed to the location: vector addition
    location.add(speed);

    // Check for bouncing
    if((location.x > width) || (location.x < 0)) {
        speed.x = speed.x * -1;  //reverse speed
    }
    if ((location.y > height) || (location.y < 0)) {
        speed.y = speed.y * -1;   //reverse speed
    }

    // Display at x,y location
    fill(175);
    ellipse(location.x,location.y,16,16);
  }
```

Below is a processing sketch for the PVector Bouncing ball. Click on the image to stop / start the animation.

8.3 PVector Object

The PVector object allows us to explore the processing object syntax. When we want to create an instance of an object, we use the object's constructor function. According to the processing PVector reference, the PVector class has

3 different constructor functions. Notice that each constructor has a unique function signature, this is an important concept called function overloading. We can have several versions of the same function, but the signature of each function must be unique. For objects, it's helpful to have different constructor functions, for the PVector, this allows it to represent both 2D or 3D vectors depending on how we initialize our instance.

```
PVector();  //default constructor function
PVector(float x, float y);  // 2 dimensional constructor function
PVector(float x, float y, float z);  //3 dimensional constructor function
```

To create a new instance of a PVector object we must use the Processing object syntax depending on which constructor we choose to use, the default constructor has no arguments, therefore the x and y properties are initialized using *dot* notation. Dot notation the syntax for calling a class's method or for setting a property value for a data element that belongs to the object's own object class. We set the x value of the location PVector instance using `location.x=100;` Note that in the code below, the object type is PVector.

```
PVector location = new PVector();  // declare a new PVector object
location.x= 100;         // initialize the x data element using dot notation
location.y= 120;         // initialize the y data element using dot notation

PVector speed = new PVector(3 , 4 );  // declare and initialize a new PVector object speed has x,y co

location.add(speed)  // use add method to add vector components of speed to location.
```

8.4 Object Dot Notation

PVector is an object and has both functions and data elements which are associated with that object. As mentioned above, we use the *dot notation* when accessing and modifying properties of an object. For PVector, x and y are properties or data elements. Below is an example of setting a property value for an instance of a PVector object.:

```
PVector location = new PVector();
location.x = 100;   // use dot notation to initialize the x property of the location PVector
location.y = 200;

location.x = location.y + 10 ; // dot notation allows accessing and resetting object properties like

ellipse(location.x, location.y, 50,50);  // use dot notation to access the x and y properties of the
```

We also use dot notation when using methods that belong to an object. Methods are functions that belong to an object so they act on that object. We use dot notation to make it clear that we want to use the object's method, rather than some global function that does not belong to the object's class. This allows the compiler to understand which function we intend to use. A common example of this would be the function the `print()` function. It is useful for debugging to be able to print some meaningful information about an object, so when we design an object class, we'll often either create a `print()`, `display()` or a `toString()` method that belongs to the class so that we can easily access and view data associated with an object. On the other hand, processing provides it's own `print()` function, so if PVector had it's own `print()` method, then the compiler would need to understand whether we intend to call the processing `print()` function or the PVector `print()` method. Dot notation syntax tells the compiler that we want to call the method associated with the object calling the function. Actually PVector has a `toString()` method, so we could use dot notation to call the method in the following way:

```
print(location.toString());              // prints '[ 100.0 , 200.0 , 0.0 ]'  which are the values fo
```

8.5 Functions: Pass by Reference

So far, when we've created functions, we have only used primitive variable types like int, float, booleans, or literal values. When these values are passed into a function, a copy of the value is passed into the function, so within the function, any modification to a value only affects the local variable. We've made the distinction between local and global variables based on the understanding that variables passed into a function are a local copy of a global variable, and so any corresponding global variable isn't changed when the local function variable is modified. Example of primitive-type function arguments: *pass-by-value*

```
float someVal=20;  // declare global variable

void doSomething(float inputVal){
        inputVal += 5;  // modify local variable
}

void doSomethingElse(float someVal){
        someVal += 10;  // modify local variable
}
void setup(){
        doSomething(someVal);   // call the function using someVal as input
        doSomethingElse(someVal);     // call the function using someVal as input

println("someVal " + someVal);  // someVal=20.0 because it is a global and a primitive-type, so pass
}
```

This is not the case when using passing objects into a function. For most cases, when we pass an object variable into a function, we actually want to have the changes take place on the object that we pass into the function. Therefore, what is passed into a function is not a copy of a variable, but is a reference or a pointer to the object. here's an example of *pass-by-reference* for an PVector object instance

```
PVector location = new PVector(20,20);   //create and initialize a PVector instance

void doSomething(PVector somePV){
        somePV.x += 5;          //modify the input PVector object x attribute;
}

void setup(){

        doSomething(location);

        println("location.x " + location.x);    // 25.0  location.x was modified due to pass-by-refe
}
```

So now that we have had experience working with the PVector object, in the next section, let's create a very simple ball class so we can create ball objects. Then we'll move on to the example in chapter 8, however where the Learning Processing book creates a car class we will make a fish class.

8.6 Questions:

1. Which of the following concepts can be represented by a PVector object? location, velocity, friction, acceleration?
2. What would be the value of location.x in the code below:

```
    PVector location=new PVector(5/2, 20);
```

3. What would be the value of location.x in the code below:

```
PVector location= new PVector( 5.0/2 , 20);
```

CHAPTER 9

Objects

In the previous section we used the PVector object to create a bouncing ball animation and learned about dot notation used by objects to access object properties and object methods. Now we're ready to create our first object, and we do this by creating a `class`. A `class` is the specification of all of the information for our object, we create a `class` to define our object's properties and methods, so we can consider that a `class` is like a recipe to make instances of objects. In Shiffman's book, he gives an analogy of the class definition as being like a cookie cutter, and a particular instance of an object being a cookie. In the code below, the Ball class would be the cookie cutter and a single ball object that's created using the constructor syntax would be like a cookie.

For our bouncing ball, we can try to come up with a set of properties and behaviors that we will use to represent our ball in our program. There are many other features of a ball that we're going to ignore as we try to come up with an abstract concept of a bouncing ball's features. We can ignore whether the ball was made by an artist, or whether it came in a box when purchased. We can ignore whether it's old and dirty, or brand new and shiny. For our simple example of a bouncing ball, we'll limit our description to the following properties and behaviors:

- **Name:** Ball
- **Properties:** size, color, location x, location y, speed x, speed y
- **Behaviors:** move, display

9.1 Ball Class

We'll continue with the last example of a bouncing ball for our first class. The syntax for a creating a class involves first naming the class, and this introduces a completely new type of syntax:

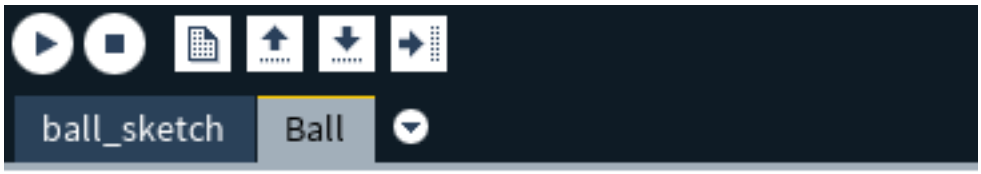
```
class Ball{

}
```

Notice there is no return type and there are no `( )`, so this is not like declaring a function. Next we need to declare the class outside of all other functions. The processing environment is setup so that we can use a new tab for creating our class code, but we could create our class on the current, main tab. It's also a convention that we use capitalization for class names. If we create a new tab, processing will prompt for the tab name, and we'll name it the same as the class name. Below we've created the code for the Ball property variables and we've also created the default constructor which has no input arguments, we have given default initialization values to the property variables:

```
class Ball{
  // Variables or Properties
  color c;
  float xpos;
  float ypos;
```

```
  float xspeed;

  //Constructor

  Ball(){  //default constructor
    c=color(255,0,0);
    xpos=width/2;
    ypos=height/2;
    xspeed=1.0;
  }

}
```

In the image below we have a new tab, named Ball, where we've created the code for the Ball class. The tab must have the same name as the class in order for us to access this class from the main ball_sketch tab where we'll have our setup() and draw() functions.




```
class Ball{
  // Variables
  color c;
  float xpos;
  float ypos;
  float xspeed;

  //Constructor

  Ball(){  //default constructor
    c=color(255,0,0);
    xpos=width/2;
    ypos=height/2;
    xspeed=1;
  }

}
```

Next we can create another constructor that allows us to set initial values for the property variables when we create the object instance. The syntax is similar to what we've used in writing other functions, except now we'll use the input parameters to initialize the class property variables, so this may seem a bit strange:

```
Ball(color tempColor, float tempXpos, float tempYPos, float tempXspeed){

        c=tempColor;   //initialize c with input tempColor
        xpos=tempXpos;
        ypos=tempYpos;
        xspeed=tempXspeed;

}
```

9.2 PVector Ball Class

Let's define the Ball class using PVectors and see what that looks like. We will need to initialize our PVector objects as part of the object constructor functions. Again here we have defined 2 different constructors.:

```
class Ball{
  // Variables
  color c;
  PVector position;
  PVector speed;
  float diameter;

  //Constructor
  Ball(){  //default constructor
    c=color(255,0,0);
    position=new PVector(width/2,height/2);  // create new PVector objects and initialize them
    speed=new PVector(3,5);
  }

  // constructor with initializations
  Ball(color tempc, float tempXpos,float tempYpos, float tempXspeed, float tempYspeed){
    c=tempc;
    position=new PVector(tempXpos,tempYpos);
    speed=new PVector(tempXspeed,tempYspeed);
  }
```

9.3 Class Methods

Then the final part of the class definition is that we need to define methods for the class, these methods are the code we'll write to create the behaviors for our ball. As mentioned in the previous section, it's convenient to create a method that will allow you to print out the values for the Ball Ojbect's property values that might be changing during a program execution, one way to do that is to create a function called `toString()` as shown below, this can be called from the processing global print() function because it takes a string as an input argument. This code would be included within the Ball class tab.

```
// class methods
  void display(){
     fill(c);
     ellipse(position.x,position.y,diameter,diameter);
  }
```

```
  void move(){
    position.add(speed);
    if(position.x > (width-diameter/2) || position.x < (0+diameter/2)){
      speed.x *= -1;
    }
    if(position.y > (height-diameter/2) || position.y <(0+diameter/2)){
      speed.y *=-1;
    }
  }

  String toString(){
    return " [ " + this.position.x + " , " + this.position.y + " ]";
  }
```

9.4 Main Sketch

Then, let's include the code that is in the main sketch page where we'll actually create our Ball object instance:

```
Ball myBall;

void setup(){
  size(300,300);
  color ballColor=color(100,200,100);
  myBall=new Ball(ballColor, 20,20,3,5);
  background(255);

}

void draw(){
   background();
   myBall.move();
   myBall.display();
   println(myBall.toString());
}

void background(){  //over-ride the processing background function to allow trails.
  fill(255,15);
  rect(0,0,width,height);  //background alpha doesn't work in processing
}
```

Below is the full code for the Ball class which includes 3 methods and 2 constructor functions and is called in the above main sketch code.:

```
class Ball{

  // Variables
  color c;
  PVector position;
  PVector speed;
  float diameter;

  //Constructor
  Ball(){  //default constructor
    c=color(255,0,0);
    position=new PVector(width/2,height/2);
    speed=new PVector(3,5);
```

```
  }

  // constructor with initialization arguments
  Ball(color tempc, float tempXpos,float tempYpos, float tempXspeed, float tempYspeed){
    c=tempc;
    position=new PVector(tempXpos,tempYpos);
    speed=new PVector(tempXspeed,tempYspeed);
  }

  // class methods
  // this method is responsible for creating the displayed ball object
  void display(){
     fill(c);
     ellipse(position.x,position.y,diameter,diameter);
  }

  //this method is responsible for determining movement of the ball
  void move(){
    position.add(speed);
    if(position.x > (width-diameter/2) || position.x < (0+diameter/2)){
      speed.x *= -1;
    }
    if(position.y > (height-diameter/2) || position.y <(0+diameter/2)){
      speed.y *=-1;
    }
  }

  // this is a convenience method to help with debugging
  String toString(){
    return " [ " + this.position.x + " , " + this.position.y + " ]";
  }

} //end of the Ball class definition
```

To review, in order to create a class, we need the following things which are all specified within a class definition:

1. Class Name
2. Instance Variables - these are properties / attributes of an object
3. Constructor functions
4. Methods, which are functions that control behavior of an object

9.5 Questions

1. Can you modify the speed attribute of a ball so that it's speed is dependent on the diameter of the ball? Smaller Balls move faster than bigger ones?
2. How can we compare to see if 2 different ball objects are equal? How would we define equal for Ball objects?

CHAPTER 10

More Objects

In this section, we'll continue to explore the concepts of classes and objects using the context of the Ball class.

10.1 this Keyword

When referring to properties and methods from within a class definition, the keyword `this` is used to refer to the actual object itself. We can use the keyword `this` to clarify some of the code we write when creating a class definition. For example, if we create 2 different class constructors, we can use the keyword with function notation: `this()`, as a way to call one constructor from within another constructor. For example, the default constructor can call another constructor using `this()` as a way to simplify the class code like below:

```
class Ball{
        color c;
        PVector position;
        PVector speed;
        float diameter;

        //default constructor uses this( ) to call the main constructor

        Ball(){
                this(color(255,0,0), 100.0, 100.0, 2.0, 2.0, 30.0);  //call the other constructor fr
        }

        // constructor that accepts input parameters

        Ball(color c, float xpos, float ypos, float xspeed, float yspeed, float diameter){
                this.position.x=xpos;
                this.position.y=ypos;
                this.speed.x=xspeed;
                this.speed.y=yspeed;
                this.diameter=diameter;
        }
}
```

10.2 Comparing Objects isEqual

So far, the methods we've written have only concerned 1 ball object. How can we write a method to allow comparison between 2 Ball objects? What would it mean for 2 unique Ball objects to be *equal*. If we try to use the same syntax that we've used to compare primitive variable values, we will have problems! With primitive variables, we can directly

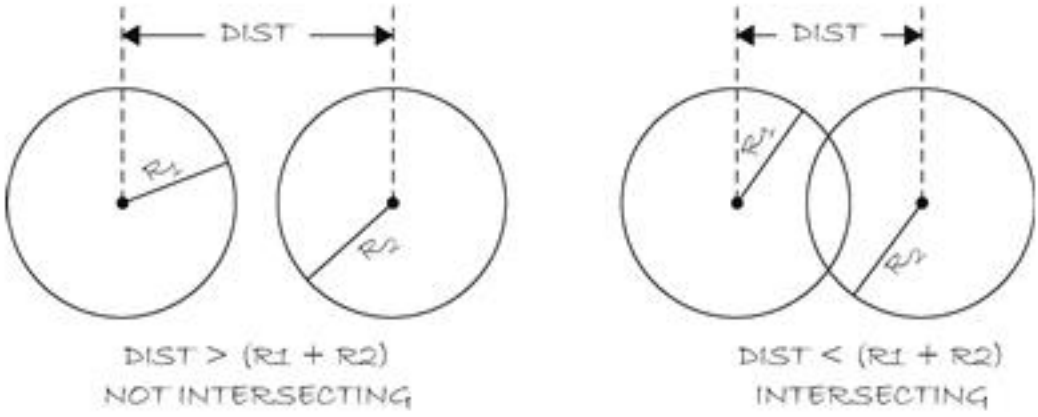
compare their values. We may need to use type-casting if we try to compare an integer with a float but the syntax would be as follows:

```
//Compare Primitive types
float float1 = 5.0;
float float2 = 4.999;
int int1 = 5;
boolean equalFloats = (float1 == float2 );  //false
boolean equalNumbers = ( int1 == float1 ); //error
boolean equalTypeCast1 = (int1 == int(float1) );  //true
boolean equalTypeCast2 = (int1 == int(float2) );  //false

// compare PVector objects
PVector vector1 = new PVector( 10, 4 );
PVector vector2 = new PVector( 5, 7 );
boolean equalVectors = ( vector1 == vector2 );  // false

vector1 = vector2;   // assignment
boolean equalVectors2 = (vector1 == vector2 );  //true, both variables point to the same memory locat

println( vector1.x )  // 5   since the variable vector1 now refers to the same objects as vector2
```

So, to continue the discussion in terms of our Ball objects, let's write a method that will allow us to check whether 2 ball objects occupy the same space on the canvas. We can look at some of the PVector methods like add(PVector pvec) to have an idea of how one object can interact with another one using methods. We'll need to use the keyword `this` in order to write our equals function. Let's agree that 2 balls are equal if they have the same size and position. Finally, our method must take a Ball as an input parameter and return a boolean as the return value:

```
boolean isEqual(Ball otherBall){
        if(this.position.x == otherBall.position.x  && this.position.y == otherBall.position.y  && th
                return true;
        }
        else
        return false;
}
```

With bouncing balls, it's unlikely that many ball objects will actually have the exact same values for position and size, so instead let's look at what collision would look like. Here we want to see if the distance between the centers of the balls is less than the sum of the 2 ball radiuses. The image below shows how distance between circle centers can be compared with circle radius size to determine if 2 circles are intersecting



10.3 Comparing Objects isIntersecting

The code below shows how we can implement this in a simple function:

```
boolean isIntersecting(Ball otherBall){
        float distance=dist(this.position.x, this.position.y, otherBall.position.x, otherBall.positio
        if( (distance <= this.diameter / 2) + (otherBall.diameter / 2)){
                return true;   //intersecting
        }
        return false   //else, no intersection so return false
}

void highlight(){   //we can call the highlight function in the draw loop to show the intersection
        this.c = color(255,255,0,80);
}
```

Here is the processing sketch.

10.4 Ball Class

Here is the full code for the Ball class that includes a test for intersection between 2 balls:

```
class Ball{

  // Variables
  color currentColor;  //current color of the ball
  color ballColor;  //store color to reset after highlighting
  color highlightColor;  //highlight color of the ball
  PVector position;
  PVector speed;
  float diameter;

  //Constructor
  Ball(){  //default constructor
    this(color(255,0,0), width/2, height/2, 3, 5 );  //call the constructor with initialization value

  }

  // constructor with initialization arguments
  Ball(color _c, float _xpos,float _ypos, float _xspeed, float _yspeed){
    currentColor=_c;
    ballColor=currentColor;
        highlightColor=color(255,255,0,40);
    position=new PVector(_xpos,_ypos);
    speed=new PVector(_xspeed,_yspeed);
  }

  // class methods
  // this method is responsible for creating the displayed ball object
  void display(){
     fill(currentColor);  //this may be highlighted or ballColor
     ellipse(position.x,position.y,diameter,diameter);
         currentColor=ballColor; //reset ballColor back to original color
  }
```

```
  //this method is responsible for determining movement of the ball
  void move(){
    position.add(speed);
    if(position.x > (width-diameter/2) || position.x < (0+diameter/2)){
      speed.x *= -1;
    }
    if(position.y > (height-diameter/2) || position.y <(0+diameter/2)){
      speed.y *=-1;
    }
  }

  //comparison method:  do comparison and return true or false

   boolean isIntersecting(Ball otherBall){
      float distance= PVector.dist(this.position, otherBall.postion);  //PVector distance between 2 p

      if( distance <= (this.diameter / 2) + (otherBall.diameter / 2)){
        return true;
      }
      return false;
  }

  void highlight(){
    this.currentColor = this.highlightColor;  //change the currentColor to be highlighted
  }

} //end of Ball class
```

10.5 Main Program Highlight Intersection

Here is the main sketch code:

```
Ball ball1;
Ball ball2;

void setup(){
  size(300,300);
  ball1=new Ball(color(100,200,100);,25,20,3,6);
  ball1.diameter=50;
  ball2=new Ball(color(255,0,0),20,20,2,4);
  ball2.diameter=40;
}

void draw(){
   background(255);

        //test to see ball1 isIntersecting ball2, highlight both if this is true:
   boolean isIntersect=ball1.isIntersecting(ball2);

   if(isIntersect){
               ball1.highlight();
               ball2.highlight();
        }
   ball1.move();
   ball1.display();
   ball2.move();
```

```
    ball2.display();
}
```

10.6 Questions:

1. Can you create a class called `Block` which creates a square shape that moves around the canvas?

CHAPTER 11

Rain Catcher Game

In Shiffman's book, he creates a very simple game based on a simple ball-like objects and a few other simple objects like a timer object and a catcher object. He uses arrays of objects so that we can use a loop to update objects for each iteration of the draw loop.

In Chapter 10, Shiffman introduces the idea of using algorithmic thinking to figure out how to design our program. So one idea is to first try to figure out the components of any program that we're trying to create. Let's follow Shiffman's project where he designs the RainCatcher game. Below he specifies the details of the game.

11.1 Rain Game Code

The object of this game is to catch raindrops before they hit the ground. Every so often (depending on the level of difficulty), a new drop falls from the top of the screen at a random horizontal location with a random vertical speed. The player must catch the raindrops with the mouse with the goal of not letting any raindrops reach the bottom of the screen

He breaks down the game program design into 4 steps:

1. Develop code for a circle object that is controlled by the mouse. This is the 'rain catcher'
2. Write a program to see if the 2 circles intersect
3. Write a timer program that executes a function every N seconds
4. Write a program with circles falling from the top of the screen.

11.2 Psudocode

Psudocode is a way to write the main concepts for the program

Setup: Initialize catcher object

Draw: Erase Background Set catcher location to mouse location Display catcher

11.3 Catcher Class Code

Here is code for the Catcher class:

```
class Catcher{
float r;  //radius
PVector position;

Catcher(){
  this(15);
}
Catcher(float _r){
        r=_r;
        position =new PVector(width/2, height/2);
}
void setposition(float _x, float _y){
        position.x=_x;
        position.y=_y;
}
void display(){
        stroke(0);
        fill(175);
        ellipse(position.x,position.y,r*2, r*2);
}
}
```

It's critical to note that in the constructor `Catcher(float r)`, we are initializing the PVector object. This is an important function of a constructor: to create any objects that are instance variables of the class. We can't use any of these objects until they've been initialized.

11.4 Ball Class Code

Here is the code for the Ball class Note that we're using PVector for speed and location:

```
     class Ball{

 // instance variables
 color currentColor;  //current color of the ball
 color ballColor;  //store color to reset after highlighting
 color highlightColor;  //highlight color of the ball
 PVector position;
 PVector speed;
 float diameter;

 //Constructor
 Ball(){  //default constructor
     this(color(255,0,0), width/2, height/2, 5, 3, 5 );  //call the constructor with initialization v
 }

 // constructor with initialization arguments
Ball(color _c, float _x,float _y, float _d, float _xspeed, float _yspeed){
   currentColor=_c;
   ballColor=currentColor;
   highlightColor=color(255,255,0,40);
   position=new PVector(_x,_y);
   speed=new PVector(_xspeed,_yspeed);
   diameter=_d;
 }

 // class methods
```

```
 // this method is responsible for creating the displayed ball object
 void display(){
    fill(currentColor);  //this may be highlighted or ballColor
    ellipse(position.x,position.y,diameter,diameter);
    currentColor=ballColor; //reset ballColor back to original color
 }

 //this method is responsible for determining movement of the ball using the PVector function ``add(
 void move(){
   position.add(speed);
   if(position.x > (width-diameter/2) || position.x < (0+diameter/2)){
     speed.x *= -1;
   }
   if(position.y > (height-diameter/2) || position.y <(0+diameter/2)){
     speed.y *=-1;
   }
 }

//comparison method:  do comparison and return true or false

  boolean isIntersecting(Ball otherBall){
     float distance= PVector.dist(this.position, otherBall.position);  //PVector distance between 2 p

     if( distance <= (this.diameter / 2) + (otherBall.diameter / 2)){
       return true;
     }
     return false;
 }

 void highlight(){
   this.currentColor = this.highlightColor;
 }

     } //end of Ball class
```

This is the end of the code for the Ball class. This class has 4 different methods. Each of these methods does a simple task. It is best to have your object methods designed to perform one well defined task. If we have a more complex task, we can break that down into simpler methods we can also call methods from within other methods if it makes our code easier to understand.

11.5 Timer Class Code

Here is the code for the timer class. It uses the processing function `millis()` which counts milliseconds since the sketch started. Shiffman uses the timer to generate an event to create a new Drop that can fall from the top of the canvas.

```
class Timer{
        int startedTime;
        int totalTime;

                        //constructors
         Timer(int _totalTime){  //constructor
                totalTime=_totalTime;
                }

                        //methods
```

```
        void start(){
        startedTime=millis();   //set the start time to the current millis value
        }

        boolean isFinished(){  //this timer determines if the timer has completed the timed interval
        int passedTime=millis()-startedTime;
        if(passedTime>totalTime){
                println("timer finished");
                return true;
        }
        else{
                return false;
        }
        }

}  //end of Timer class
```

11.6 Object Inheritance

Here, we are going to use *Object Inheritance* is the code for the Drop class, it is a child class of the Ball class and it inherits the instance variables and methods from the Ball class. we use the `super` keyword to refer to methods in the parent Ball class:

```
class Drop extends Ball{
 boolean isActive;  //this is instance variable for drop class
 color dropColor;

 Drop(){
   this(random(width), -10);
 }

 Drop(float _x, float _y){
   // call the Ball constructor
   super();
   this.position.x=_x;
   this.position.y=_y;
   this.diameter=5;
   this.speed.x=0;
   this.speed.y=3;
   dropColor=color(0,50,255,100);
   this.ballColor=dropColor;
   isActive=true;
 }

 void move(){
   if(isActive){
      position.add(speed);  //we've set x speed to 0;
      if(position.y>=height+10){
        isActive=false;
    }
   }
 }

 void display(){
   super.display();
 }
```

```
}
```

In the above code, we have created a class that's a child class of the Ball class. We have used the keyword `super` within the constructor so that we're calling the constructor for the `Ball` class. We have used the `extends` keyword in the first line of the class declaration to show that this class is a child class of the `Ball` class. Any Drop object has access to the methods and instance parameters of the `Ball` class. Since the `Drop` class has it's own `move()` method, then when a Drop object calls the `move()` method, it is this version that will be executed.

11.7 The Main Program

Here is a start of a main program where we are testing each of our classes. It's important to keep straight the fact that we're declaring our classes in separate tabs, but all of the code to execute the program is all contained in the first processing tab. In that tab, we have our processing setup function and the draw function. As we've done before, we declare any global variables above and outside of the `setup()` and `draw()` functions. These are object variables so we use the Class name, then the name of the object instance to declare the global object

class name: `Catcher`

object instance name: `myCatcher`

Here's the code for executing the beginning of our game:

```
      //rain catcher game: main file
      Catcher myCatcher;   //declare a Catcher object named myCatcher
      Ball ball1;
      Timer timer1;
      Drop drop1;

      void setup(){
              size(300,300);
              myCatcher=new Catcher();  // initialize using the Catcher default constructor
              timer1=new Timer(2000);  // initialize a Timer object
              timer1.start();    //call the start( ) method
              ball1=new Ball(color(0,255,100),15,25,20,3,8);
              smooth();
              drop1=new Drop(14,5);   //initialize drop1 using the Drop constructor

      }

      void draw(){
              background(255);
              myCatcher.setposition(mouseX, mouseY);  //
              myCatcher.display();
              drop1.move();
              drop1.display();
              ball1.move();
              ball1.display();
              if(timer1.isFinished()){
              timer1.start();  //reset the timer when it is finished
              }
}
```

In the code above, the first thing we determine is the location of the catcher object based on the user's mouse position. Then we display the myCatcher object. Similarly, with the drop1 and ball1 objects, first we move the objects, then we display the objects.

So far, we have several objects moving on the screen, but we need to re-factor this code in order to make some type of a game. We'll want to have lots of drop objects moving on the canvas. Also, let's make use a paddle object instead of Shiffman's catcher object. The paddle object will be controlled by keyboard movement, then collisions will be determined based on whether a falling drop object intersects with the paddle. We'll cover this in the next section.

11.8 Test Driven Development

Below is an example of the program, here we're just testing the code for each object that we've created. It's a good idea to create your code in an incremental manner, so that you can discover errors early on. For each section of code that you create, identify some way that you can test whether your code is functioning correctly before moving on to create new code. This is the idea behind test-driven development (TDD), where you would create some series of tests for each section of code, to insure it's working correctly, and the code for the tests is actually written before the code that you will be testing.

11.9 Questions:

1. How can we test whether the method `isIntersecting( )` works correctly?
2. How can we test whether the timer object is working correctly?

CHAPTER 12

Project 2a

Khiem Le, created 7/10/5, updated 7/19/15

Based on Shiffman's RainDropGame, and on the program code for Learning Processing: exercise 10.4, we want to build a slightly more advanced game based on this program.

1. We will add a start and stop button to enable the player to start and stop the game. The buttons are labeled Start and Stop respectively.
2. We will have two varieties of falling objects: regular drops and tiny drops. Tiny drops are smaller than regular drops and drop faster, with a more unpredictable trajectory, but catching a tiny drop will earn extra bonus points.
3. The score will still be displayed after the game is over

You are required to implement:

- a Game object, which uses other objects
- the regular and tiny drops as child classes of a parent "Drop" class
- the labeled buttons as child classes of a parent "Button" class

12.1 Game class

To begin design of our game, let's look at the changes we'll need to make, to modify Shiffman's exercise 10.4 project code. In Shiffman's project, he has all of the game type code simply implemented within the processing draw loop.

We'd like to re-factor his program and make a separate Game class in order to make it easier to organize and understand our program code. If we look at his Draw loop code, we can see that he's incrementing a score-type variable, he's keeping track of misses, etc. So, let's create a Game Class with the following instance variables. Some of them are the game related variables from exercise 10-4 moved inside the Game class.

12.1.1 Game: Instance Variables

- score: `int` // current score
- level:`int` // level the game is currently in
- state: `int` // Game can be in inactive or active state
- numberDropsDone: `int`// a drop is done if it is caught by the catcher or if it reaches the bottom of the canvas (this is called levelCounter in 10.4)

- numberLivesLeft: `int`// counter value initialized at MAX_NUMBER_LIVES and decremented when a drop reaches the bottom. (this is called lives in 10.4)
- totalDrops: `int` // number of drops generated
- gameOver: `boolean`// set to true when a game is over
- Wbar: `float`// width of the background bar behind the buttons
- startBtn: `LabeledButton` // Button to start the game
- stopBtn: `LabeledButton` // Button to stop the game
- catcher:`Catcher`// Catcher class
- timer: `Timer`// Timer class
- drops: array of drop objects // size of array is MAX_NUMBER_DROPS

We can use integer values to indicate the possible game state values, this will allow us to use a switch(gameState) structure. We can use the keyword `final` to make it clear that these are values that shouldn't be changed within the program

```
final int INACTIVE=0;
final int ACTIVE=1;
```

In a similar fashion:

```
final int MAX_NUMBER_DROPS = 50;
final int MAX_NUMBER_LIVES = 10;
```

Next, we'll want a few methods:

12.1.2 Game: Methods

- start(): `void` // initializes the proper Game variables when the game starts
- drawButtons(): : `void`// draws the startBtn and stopBtn buttons by invoking the drawButton() methods of the two LabeledButtons
- displayScore: `void` // displays score, even when game is over. Score is reset when a new game starts
- displayLevel(): `void`
- drawBar(): `void` // draws the background bar behind the buttons
- displayGameOver(): `void` // displays the "Game Over" message. This is essentially copy and paste of the block code under if(gameOver) in 10.4's draw loop
- play(): `void`// generates and displays the drops, tracks the catcher, updates the game variables. This is essentially copy and paste of the remaining part of the draw loop from 10.4

12.2 Button Objects

If we want to include some Button objects in our game, like a startButton and a stopButton, then we'll need to look at what a Button object would look like. Shiffman provides an example of a Button object in {{*Exercise 9.8*}}_ Our buttons will be created from the "LabeledButton" class, a child class of the "Button" class. Unlike the "Button" class in Shiffman, our "Button" class does not have an "on" variable, but it has a "col" variable of type color. We will refer to our "labeled button" as "button". To integrate a button in the Game class, it's important to remember to create and initialize the buttons in the Game constructor by invoking the constructor of the buttons with the new clause

```
Game(){
        score = 0;
        level = 1;
        state = INACTIVE;
        Wbar = width/7;
        startBtn = new LabeledButton(width - Wbar + 5, 10, 40, 30, colYellow, "Start");
        stopBtn = new LabeledButton(width - Wbar + 5, 50, 40, 30, colYellow, "Stop");
        . . .
}
```

To take startBtn as an example,

- width-Wbar + 5 is the x coordinate of the button
- 10 is the y coordinate of the button
- 40 is the width of the button
- 30 is the height of the button
- colYellow is the normal color of the button
- "Start" is a String used to label the button when it is displayed

12.3 LabeledButton class

The LabeledButton class inherits all the variables of the Button class, namely:

- X: `float` // x coordinate of button
- Y: `float` // y coordinate of button
- W: `float`// width of button
- H: `float` // height of button
- col: `color` // color of button

The child has in addition the label variable

- label: `String` // Button label

The LabeledButton class inherits all the methods of the Button class, namely:

- rollOver(x, y): `boolean` // returns true if the point (x,y) is over the button
- drawButton(): `void` // displays the button with the appropriate color (highlighted color if the mouse is over the button, normal color otherwise)

The LabeledButton has an overriding drawButton() method which displays the button and the label on top.

12.4 Use of objects by the Game class

The Game class uses several objects, namely: startBtn, stopBtn, catcher, timer and drops. These objects are members of the Game class. Each of these member objects must be created and initialized in the Game constructor by invoking the constructor of the corresponding classes with the "new" clause. A more complete version of the Game constructor with these invoked constructors is shown below

```
Game() {
        score = 0;
        level = 1;
        state = INACTIVE;
        Wbar = width/7;
        startBtn = new LabeledButton(width - Wbar + 5, 10, 40, 30, colYellow, "Start");
        stopBtn = new LabeledButton(width - Wbar + 5, 50, 40, 30, colYellow, "Stop");
        catcher = new Catcher(32); // Create the catcher with a radius of 32
        numberDropsDone = 0;
        numberLivesLeft = MAX_NUMBER_LIVES;
        totalDrops = 0;
        drops = new Drop[MAX_NUMBER_DROPS];    // Create spots in the array
        timer = new Timer(300);   // Create a timer and set initial value to 300 milliseconds
        gameOver = false;
  }
```

Additionally, to refer to a member of an object which is member of the Game class, you may have to use multiple levels of dots. For example, in the main tab, only the myGame object is known. To refer to the rollOver() member method of the startBtn member of myGame, you have to refer first to the startBtn member, then to the rollOver() member of startBtn, and that is denoted myGame.startBtn.rollOver().

12.5 Game Button Integration

Now we need to figure out how to integrate the button event handler into the game. We will use the MouseClicked( ) event.:

```
//this code is in the main program tab

void mouseClicked() {
        switch(myGame.state) {
        case INACTIVE:
                if (myGame.startBtn.rollOver(mouseX, mouseY)) { // Start the game if the player clic
                myGame.gameOver = false;
                myGame.start();
                }
                break;
        case ACTIVE:
                if (myGame.stopBtn.rollOver(mouseX, mouseY)) { // Stops the game if the player click
                myGame.gameOver = true;
                myGame.state = INACTIVE;
                }
        }
}
```

12.6 Game Instance

All of the above code assumes that we will define, initialize and utilize an object of the Game class in the main program tab. Since we need access in the draw loop, as usual, we'll declare the object above the setup function, initialize in the setup function, and then use in the draw loop:

```
//this code is in the main tab

//other global variables
Game myGame;
```

```
void setup(){
//other initializations

myGame= new Game();   //call the Game constructor, here we call the default constructor
}  //end setup

void draw() {
background(255);
switch(myGame.state) {
case INACTIVE: // Game inactive
        if (myGame.gameOver)
                myGame.displayGameOver(); // Display the "Game Over" message
        break;
case ACTIVE: // Playing
        myGame.play();
        break;
} // end switch
myGame.drawBar();
myGame.drawButtons();
myGame.displayScore();
myGame.displayLevel();
}
```

12.7 Inheritance

12.7.1 Child classes of the Drop Class

The next modification for our game is that we're going to use different drop types: regular drops and tiny drops. Because they are both very similar to the drop class in 10.4, we will use inheritance and define them as child classes of drop.

12.7.2 Adaptation of Drop

To the Drop class in 10.4, we add a "score" variable and a getScore() method.

12.7.3 RegularDrop

RegularDrop is a child of the Drop in 10.4.:

```
class RegularDrop extends Drop {
          int score;

          RegularDrop() {
                super();
                score = 1;
          }

          int getScore() {
                return score;
          }
        }
```

12.7.4 TinyDrop

TinyDrop is also a child of the Drop in 10.4.Because TinyDrop moves differently than a drop, we define an overriding move() method. We also have a getScore() method.

When one class inherits from another class, any method that is not specified in the child class, will be implemented using the method in the parent class. This is called *Method Overriding*, and it means that when a child class has code that implements the same method that's also in parent class, then it is the child method code which is executed, if a child object calls that method. In essence, we end up with 2 different versions of one method, each with the same function signature, but with different code within the function body. So we need to understand the rules the compiler uses when determining which method to execute.

So, to summarize, when an object from a child class executes a method call, the compiler first looks in the class definition for the child object to see if that method is implemented in the child class, if so, then that's the version that is executed. This a major benefit of using inheritance, we only need to make changes to methods or features that are different in the child class.

```
//this code is in the TinyDrop Class tab


class TinyDrop extends Drop {
        int score;

        TinyDrop() {
                super();
                r = 6;                  // TinyDrops have smaller size
                speed = random(5, 7);   // Pick a higher random speed
                c = color(150, 100, 150); // Color
                score = 2;
        }
        void move() {
                // Increment by speed
                y += speed;
                x += random(-3, 3); // unpredictable fluctuation in trajectory
        }

        int getScore() {
                return score; // Score 2 points instead of 1, if catch TinyDrop
        }
}
```

12.7.5 Making Drops

In Shiffman's game, there are several important distinctions we need to think about, which control the structure and behavior of our game, in Shiffman's game, this structure is created in the main program tab. The general idea is that he has an array: drops[] that stores the Drop objects, we'll modify this so that it can also contain Drop sub-class objects like RegularDrop and TinyDrop.

1. `Drop[] drops;` //declares an array of Drop objects
2. `drops = new Drop[50];` //initializes the array to a size of 50 elements
3. **`if(timer.isFinished()){ .... }` //inside this block of code is where new drops are** actually created each time the timer goes off.
4. **inside the block: `if(timer.isFinished()){ ... }` is where we need to figure out how** to create different types of drops

So, Let's start by focusing inside the block of code where the `timer.isFinished()` has evaluated to true:

```
if (timer.isFinished()) {
  // Deal with raindrops
  // Initialize one drop
  if (totalDrops < drops.length) {
                drops[totalDrops] = new Drop();//(game.levels[game.currentLevel].dropSpeed);
                // Increment totalDrops
                totalDrops++;
  }
  timer.start();
}
```

In the above code, only 1 drop is created each time the timer goes off! This drop is created in the array location: `drops[totalDrops]` The first time a drop is created, it's in the first array positon: `drops[0]` After the drop is created, totalDrops is incremented to `1`: `totalDrops++` So, the next time the `timer.isFinished()` is true, then the next drop will be created in the array location: drops[1]. For our game, we want to create different types of drops so we'll take advantage of the idea that inheritance allows us to use *Polymorphism*

12.8 Polymorphism

As discussed above, we used inheritance to extend the Drop class, we created two child classes: RegularDrop and TinyDrop. So, the beauty of this is that we can now put child objects in an array of that has been declared to contain Drop objects. This is a manifestation of polymorphism, it means that a parent class 'reference' can be used to refer to a sub-class object. So, we can do the following:

```
Drop someDrop = new TinyDrop();     //someDrop is a Drop reference, it points to a TinyDrop object.
```

This might not seem like a very important feature on initial inspection, however, it is one of the powerful features that result from the Object-Oriented concept of Class Inheritance. So, now we can change the game code so that when the timer goes off, we can create RegularDrop or TinyDrop objects instead of Drop objects:

```
drops[totalDrops] = new TinyDrop();
//or

drops[totalDrops] = new RegularDrop();
```

To make our game interesting, we want to choose randomly between TinyDrop and RegularDrop. In addition, we want to be able to adjust the statistical percentage of RegularDrop vs. TinyDrop. This can be achieved with the following code:

```
if (random(100) < PERCENT_REGULAR) {
  drops[totalDrops] = new RegularDrop();
} else {
  drops[totalDrops] = new TinyDrop();
}
```

random(100) will return a float value randomly chosen between 0 and 100. PERCENT_REGULAR is the knob variable to control the percentage of generated drops which are RegularDrop. For example, if PERCENT_REGULAR is 90, 90% of the drops will be RegularDrop, and 10% will be TinyDrop. PERCENT_REGULAR can be added as a variable of the Game class.

```
float PERCENT_REGULAR = 90;
```

12.9 Scoring

In the original code of 10.4 shown below, whenever the player catches a drop, the score is incremented by one.:

```
if (catcher.intersect(drops[i])) {
  drops[i].finished();
  levelCounter++;
  score++;
}
```

In our project, we want to increment the score by one if the player catches a RegularDrop and increment by 2 if the player catches a TinyDrop. To achieve that, we add a "score" variable, along with a getScore() method to the parent Drop. The "score" variable is initialized to 1 and 2 for the RegularDrop and TinyDrop respectively. When the player catches a Drop, instead of incrementing the score by 1, we increment it by getScore().:

```
if (catcher.intersect(drops[i])) {
  drops[i].finished();
  numberDropsDone++;
  myGame.score += drops[i].getScore();
}
```

12.10 Other classes

The Timer, Catcher, Drop classes are as defined in 10.4.

CHAPTER 13

Paddle Drop Game

Based on Shiffman's RainDropGame, and on the program code for Learning Processing: exercise 10.4, we want to build a slightly more complicated game based on this program. Our game will use a paddle object to try and catch the falling drops. We'll also use *Object Inheritance* to have several different objects that fall during our game, and we'll use the PShape object to allow the use of .svg images for interesting drop objects. Also, we'll use a Game class for methods and variables related to the game state, this will include Button class to start the game and to allow the game to be reset. A more advanced version of the game could also include gameLevels and other more advanced features

13.1 Planning

To begin design of our game, let's look at the changes we'll need to make, to modify Shiffman's exercise 10.4 project code. In Shiffman's project, he has all of the game type code simply implemented within the processing draw loop. We'd like to re-factor his program and make a separate Game class in order to make it easier to organize and understand our program code. If we look at his Draw loop code, we can see that he's incrementing a score-type variable, he's keeping track of misses, etc. So, let's create a Game Class, initially we can begin with the following instance variables, we will probably expand on these at some point:

13.1.1 Game: Instance Variables

- score: `int`
- gameState: `int`
- missedCount: `int`
- maxAllowedMisses: `int`
- startBtn: `button`
- resetBtn: `button`

We can use integer values to indicate the possible gameState values, this will allow us to use a switch(gameState) structure. We can use the keyword `final` to make it clear that these are values that shouldn't be changed within the program

```
final int START=0;
final int ACTIVE=1;
final int END=2;
```

Next, we'll want a few methods, we may add more methods later, but we can immediately imagine we'll need the following methods:

13.1.2 Game: Methods

- display( ) : `void`
- reset( ) : `void`
- isGameOver( ) : `boolean`

13.2 Game Button Objects

If we want to include some Button objects in our game, like a startButton and a resetButton, then we'll need to look at what a Button object would look like. Shiffman provides an example of a Button object in Exercise 9.8 To integrate the a start button in the Game class, it's important to remember to initialize the button in the Game constructor, aslo, for the simple button class and click methods that we've created, we want to make sure out buttons don't have the same position values because we are using the position to determine if the button has been clicked:

```
Game(){
        int gameState=START;
        int score=0;
        startBtn=new Button(width/2, height/2+20,70,50);  //make sure buttons aren't in the same loca
        resetBtn=new Button(width/2, height/2-50,70,50);
        }
```

13.3 Game Display - Buttons

Then, in the Display method for the game, we can display the buttons depending on what the gameState is:

```
//this code is in the Game class tab

void display(){
        switch(gameState){
          case 0:  //Game is in 'Start' mode - show start button
                fill(255);
                rect(0,0,width,height);
                startBtn.display();   //here's where we display the start button
                fill(0);
                text("Push to Start",width/2,height/2-10);
                break;
         case 1:  //game is in 'ACTIVE' mode
                fill(100,150,200);
                rect(0,0,width,height);
                fill(0);
                stroke(0);
                text("Score: " + score,20,20);
                break;
          case 2:   //game is in 'END' mode
                resetBtn.display();   //here's where we display the reset button
                fill(0);
                text("Push to Restart",width/2,height/2-10);
                break;
                }
        }
```

13.4 Button State

The button objects have a display method which we've called above in the game.display( ) method. So, the button knows what to look like based on it's internal state. That is the button's responsibility. However the main reason that we want to use a button is so we can change something in the program whenever someone clicks on the button and changes the button's internal state. So, the Button objects have an internal state that is boolean: on, which is either true or false. This state is changed when the button.click() method is called:

```
      /// this code is in the Button Class:: notice that we'll have a problem if the buttons have the
      /// which might not be obvious if they're displayed at different times.

      void click(int mx, int my) {  //input is mouseX, mouseY
              // Check to see if a point is inside the rectangle
              if (mx > x && mx < x + w && my > y && my < y + h) {
              on = !on;
  }
}
```

13.5 Game Button Integration

Now we need to figure out how to integrate the button event handler into the game. So, if we look back at the main program code, we have a MouseClicked( ) event and this is where the game.startButton.click() code must be located so it's executed when the user clicks the mouse. There are 2 different types of game methods called here, first is the Button click() method for each button. The second is game1.checkState( ). It might not be obvious that we would want to have this type of method, but it makes it easier within the Game objects to determine what the impact of the button clicks has on the game:

```
//this code is in the main program tab

void mouseClicked(){
          game1.resetBtn.click(mouseX,mouseY);
          game1.startBtn.click(mouseX,mouseY);
          game1.checkState();    /// this helps us determine what to do when there are multiple butto
}
```

So, now we need to write the code for this checkState( ) method within the Game Class. So, first thing we do is see if the button states have been activated so that 'on' == true. If so, then we want to use this as a trigger to change the game state. However, it's important to remember to set the button `on` state back to false. See the code below:

```
  //this code is in the Game class tab

      void checkState(){
              if(startBtn.on==true){
                      gameState=ACTIVE;   //change game to active state
                      startBtn.on=false;
              }
              if(resetBtn.on==true){
                      gameState=START;   //change game to start screen
                      resetBtn.on=false;
              }
}
```

13.6 Game Instance

All of the above code assumes that we will define, initialize and utilize a Game object in the main program tab. Since we need access in the draw loop, as usual, we'll declare the object above the setup function, initialize in the setup function, and then use in the draw loop:

```
//this code is in the main tab

//other global variables
Game game1;

void setup(){
//other initializations

game1= new Game();   //call the Game constructor, here we call the default constructor
}  //end setup

void draw(){   //here is how we might use this code

        game1.display();

        if(game1.gameState==game1.ACTIVE){ // put most active game code in here
                if (timer.isFinished()) {
                if (totalDrops < drops.length) {
                drops[totalDrops] = new Drop();
                totalDrops++;
                }  // end if
                timer.start();
                }  //end timer
                for(int i=0;i<totalDrops;i++){
                if(drops[i].isActive==true){  //only look at active drops
                   boolean isHit=false;
                   drops[i].move();
                   isHit= drops[i].isIntersecting(paddle1);
                        if(isHit){
                                println("isHit");
                                drops[i].isActive=false;
                                drops[i].y= height + drops[i].getBottomY();  //move off screen
                                game1.score++;
                   }
                drops[i].display();
                }  //end isActive
        }//end drop[] loop
        paddle1.display();
        }  //end game1.Active

}  //end draw
```

13.7 Inheritance

13.7.1 Child classes of the Drop Class

The next modification for our game is that we're going to use several different drop types. So, we'll have 2 different classes that inherit from the Drop class. Let's say we'll have SeaHorses and Stars. Since the behavior of these objects

will be almost identical to Shiffman's Drop class, it makes sense for us to use object inheritance when defining these objects.

13.7.2 Method Override

It's obvious that these objects will have unique display() methods which display their unique shapes. Other methods like move() might be identical to the Drop method: move( ).

When one class inherits from another class, any method that is not specified in the child class, will be implemented using the method in the parent class. This is called :ref: *Method Override*, and it means that when a child class has code that implements the same method that's also in parent class, then it is the child method code which is executed, if a child object calls that method. In essence, we end up with 2 different versions of one method, each with the same function signature, but with different code within the function body. So we need to understand the rules the compiler uses when determining which method to execute. So, to summarize, when an object from a child class executes a method call, the compiler first looks in the class definition for the child object to see if that method is implemented in the child class, if so, then that's the version that is executed. This a major benefit of using inheritance, we only need to make changes to methods or features that are different in the child class.

For now we can simply create a Seahorse class that inherits from the Drop class using the code class code below:

```
//this code is in the Seahorse Class tab

class Seahorse extends Drop{
                PShape s;
                float sWidth;
                float sHeight;
                float bottomY;
                float bottomX;

                Seahorse(){
                        super();   //call the Drop constructor
                }

                void display(){
                    //some code to display the seahorse which is different than a drop
                }
}
```

13.7.3 Making Drops

In Shiffman's game, there are several important distinctions we need to think about, which control the structure and behavior of our game, in Shiffman's game, this structure is created in the main program tab. The general idea is that he has an array: drops[] that stores the Drop objects, we'll modify this so that it can also contain Drop sub-class objects like Stars or Seahorses.

1. `Drop[] drops;` //declares an array of Drop objects
2. `drops = new Drop[50];` //initializes the array to a size of 50 elements
3. **`if(timer.isFinished()){ .... }` //inside this block of code is where new drops are** actually created each time the timer goes off.
4. **inside the block: `if(timer.isFinished()){ ... }` is where we need to figure out how** to create different types of drops

So, Let's start by focusing inside the block of code where the `timer.isFinished()` has evaluated to true:

```
        if (timer.isFinished()) {
  // Deal with raindrops
  // Initialize one drop
  if (totalDrops < drops.length) {
                drops[totalDrops] = new Drop();//(game.levels[game.currentLevel].dropSpeed);
                // Increment totalDrops
                totalDrops++;
  }
  timer.start();
}
```

In the above code, only 1 drop is created each time the timer goes off! This drop is created in the array location: `drops[totalDrops]` The first time a drop is created, it's in the first array positon: `drops[0]` After the drop is created, totalDrops is incremented to `1`: `totalDrops++` So, the next time the `timer.isFinished()` is true, then the next drop will be created in the array location: drops[1]. For our game, we want to create different types of drops so we'll take advantage of the idea that inheritance allows us to use *Polymorphism*

13.8 Polymorphism

As discussed above, we used inheritance to extend the Drop class, we created a child class: Seahorse. So, the beauty of this is that we can now put Seahorse objects in an array of that has been declared to contain Drop objects. This is polymorphism, it means that a parent class 'reference' can be used to refer to a sub-class object. So, we can do the following:

```
Drop someDrop = new Seahorse();    //someDrop is a Drop reference, it points to a Seahorse object.
```

This might not seem like a very important feature on initial inspection, however, it is one of the powerful features that result from the Object-Oriented concept of Class Inheritance. So, now we can change the game code so that when the timer goes off, we can create Seahorse objects instead of Drop objects:

```
drops[totalDrops] = new Seahorse();
```

This is a start, but in addition, to make our game interesting, we want to have a variety of drop type objects falling in the game, yet we only want to create 1 drop each time the timer goes off. So we need some way to control this. We're looking for random behavior, like flipping a coin. Also, using a switch statement will allow us to easily add more types of dropping objects without making big changes to the code. The switch statement can't use a randomly generated float value, it needs an integer. So, first thing we need to generate a random integer and we can do this by type casting the random number to an integer. This will result in truncation of the integer value, so if we used Random(0,1), we'd only ever get the integer `0` as a generated value. So, the code below generates 2 different values, 0 and 1, then we can use that to randomly generate different types of drops.:

```
int choice = (int) random(0,2);  //  gives 0,1 vaules

switch(choice){

case 0:  drops[totalDrops]= new Seahorse();
                break;

case 1:  drops[totalDrops] = new Star();
                break;
}
```

The code as Shiffman has written it, means that there will never be more than 50 drops created, in other words, once totalDrops>=50, no new drops will be created. It would not be much more difficult to have additional logic to have an else{ } block where, once we know the entire array has been filled with drops, to continue the game, we could just

loop through the array and find inActive drops and then create new drops in those spots. Below I have just pasted in the duplicated code from above, this would not be the most elegant or efficient way to do this but it should at least convey the idea of how this might be implemented:

```
   if(game1.state==game1.ACTIVE){
           if(totalDrops < drops.length){
     int choice = (int)random(0,2);  //  gives 0,1 vaules
        println("choice " + choice);
        switch(choice){
           case 0:  drops[totalDrops]= new Seahorse();
                     break;
           case 1:  drops[totalDrops] = new Drop();
                     break;
         }    //end switch
totalDrops++;
    }  //end if
     else{  //the array is already full of drops, some are not active, find one isFinished and create
           for(int i=0; i< drops.length, i++){
                    if(drops[i].isFinished){   //or !drops[i].isActive
                                    int choice = (int) random(0,2);  //  gives 0,1 vaules
                                    switch(choice){
                                    case 0:
                                                    drops[i]= new Seahorse();
                                                    i=drops.length; //  since we've found one, drop ou
                                                    break;
                                    case 1:
                                                    drops[i] = new Star();
                                                    i=drops.length;
                                                    break;
                                                    }   //end switch
                    }  //end if
           }  //end for
     }  //end else

           ///the rest of the game1.ACTIVE code

           }
```

Questions:

CHAPTER 14

Paddle Object

In the previous section, we reviewed Daniel Shiffman's Rain Game, Object-Oriented Game. Now, we want to customize the game to make it a bit more interesting. The next change we'll make is to add a paddle that can catch or hit the falling objects. If we allow the objects to bounce, then that could provide a user with extra scoring opportunities. For now, let's just look at how we can implement a paddle that moves left and right in response to keyboard input.

14.1 Pong Game

Dr Doane, in his online article: *Thinking Through a Basic Pong Game in Processing* provides a nice tutorial on how to create a pong game using processing. While it's not an object-oriented approach, it still provides a very good overview and details of ideas we'll want to implement. To start with, he discusses how we can use the processing keyboard functions to control movement of the paddle. Just as with mouseEvents, when a user presses a key, we can use that input to allow interaction with our program.

The initial code in Dr Doane's tutorial describes the motion of a ball as it bounces against the edges of the canvas. The main idea is that there are boundaries of the canvas where we need to test to see if the ball has reached those boundries and if it has, then we need to change the direction of the ball object's speed. In an object oriented approach, this behavior would be implemented in the Ball class definition, in the `move( )` method.

14.2 KeyPressed Event

Dr. Doane then refers to the processing reference code in order to determine how to move the paddle object in response to a user's keyboard interaction. Below is the processing example code:

```
// based on code from http://processing.org/reference/keyCode.html

void keyPressed() {
        if (key == CODED) {
                if (keyCode == UP) {
                paddleY = paddleY - 30;
        } else if (keyCode == DOWN) {
                paddleY = paddleY + 30;
                }
        }
}
```

In the code above, the first thing is to note that we want to know if the user has interacted with our program using the keyboard. If that's happened, then a keyPressed event is triggered. Similar to mousePressed events Processing provides a function keyPressed( ) that is triggered when the user interacts with the keyboard. Then, within the keyPressed( )

function, we need to determine how we want our program to respond to the keyPressed event. The keyPressed event stores the a key value, and it remembers the most recent key that has been pressed. For special keys like arrow keys, we need to also use the keyCode values, so we can tell if the key that was most recently pressed corresponds to a special key, the arrow keys. In the code above, keyCode == UP, is used to determine whether to move the paddle upwards.

For our project, we'll be using a paddle that moves horizontally, so we'll look at whether keyCode == LEFT, or KeyCode == RIGHT, and then we'll need to create code that changes the behavior of our paddle's movement based on these keyCode comparisons.

14.3 KeyPressed Event Handlers

First we need to create a Paddle class: This will be simliar to the Ball class, but we'll have a rectangular object that moves based on the users keyboard interactions. So, instead of the move() method, we'll have pressedLeft() and pressedRight() methods:

```
   //this code is part of the Paddle class definition

   void pressedLeft(){
  if(x>0){   //check to make sure that the paddle doesn't move off the left edge
     x=x-speed;  // decrease x position to move the paddle left
   }
 }
 void pressedRight(){
   if(x+pWidth<width){  //make sure paddle stays within the right canvas border
           x=x+speed;
   }
}
```

The other methods and constructors are basically just like the Ball object, where we have paddle position coordinates: x,y and paddle dimensions pWidth, pHeight. We also have a speed variable that controls how fast the paddle moves.

14.4 KeyPressed Paddle Method Calls

For our program, we'll actually want to use these pressedLeft() and pressedRight() methods within the keyPressed event. The pressedLeft( ) method is an event handler. It's code that we want to be executed when the keyPressed event occurs. So, in the main program, we would create a Paddle object, for example paddle1. Then in the keyPressed event, we'd use the paddle1 object to call it's pressedLeft( ) method as in the code below:

```
// This code is in the main program, below the draw() function

void keyPressed() {
        if (key == CODED) {
                if (keyCode == LEFT) {
                        paddle1.pressedLeft( );
                }
        else if (keyCode == RIGHT {
                        paddle1.pressedRight( );
                }
        }
}
```

14.5 Arrows: State Indicators

The example below displays left and right arrows when the user presses the arrow keys. In order to display the correct arrow, I've created some additional variables as part of the paddle class, these are *state* variables that keep track of the last keyPress event. I'm using `int` variables, since I want to have 3 possible values: pLEFT, pRIGHT, pNONE (which is the starting position).

Click inside the sketch to activate, then use the right and left arrows to move the paddle.

14.6 Final Keyword - Constant Values

This introduces 4 new instance variables in order to keep track of and display the red arrows which indicate direction, Note the use of the `final` keyword:

```
 // new instance variables for the Paddle class

int direction;  //this variable stores the current direction
final int pNONE=0;   //initial direction state variable
final int pLEFT=1;   // left direction state variable
final int pRIGHT=2;  //right direction state variable
```

The `final` keyword is used to indicate that this value should not be ever be changed, these values are used as 'constants' within the program. The use of capital letters also indicates that these are special values which are constants and shouldn't be modified in the program. The constants are used to set the value of direction, the use of `int` makes it easy to use a switch statement for our program logic. In the `display()` method of the Paddle class, we use the switch statement to determine which arrow method to call. Note that we've created separate display functions for each arrow within the Paddle class, this makes our code logic easier to understand. Below is part of the `display()` code for the Paddle class, showing how we've used switch to control which arrow is displayed:

```
// this code is in the Paddle class: display() method

switch(direction){   //test the current value of direction
                case(pNONE):       //if the initial value, do nothing
                break;
        case(pLEFT):       //if pLEFT, display left arrow
                this.displayLeftArrow();   // call this Paddle method
                break;
        case(pRIGHT):     //if pRIGHT, display right arrow
                this.displayRightArrow();   // call this Paddle method
                break;
}
```

14.7 Set the State Variable

So, next we need to figure out *where* to change the value of direction. We have already created the Paddle methods: `pressedLeft()` and `pressedRight()`, and we know these methods are executed when the user presses the left or right keyboard arrows, these Paddle methods are *event handlers* that we have created, and they are executed in the global `keyPressed( )` event by a Paddle object. So, it makes sense that we would want to change the direction state variable when this event occurs, and we'll want to do that within the Paddle class itself, because a paddle object should be responsible for knowing what behaviors need to occur when the Paddle method: `pressedLeft()` event handler is executed. Below is the new code:

```
    // this code is in the Paddle class: pressedLeft() method

    void pressedLeft(){
           if(x>0){
                   x=x-speed;
           direction=pLEFT;  //here we set the direction state value to pLEFT
     }
}
```

14.8 Intersection

In Shiffman's game, both of his objects are circular so that has made testing for intersection much easier. In our game, we're going to use a rectangular paddle and .svg PShape objects. Both of these elements have their x,y locations at the upper left corner of the object, whereas circles have x,y defined at the center. However, our paddle can't move in the y direction, so that makes it a little easier to check for intersection.

After noticing some weird behavior when implementing the isIntersecting within the Paddle class, I have decided to move the code to the Drop classes. So, we'll pass in a Paddle object, and call the method using the drop[i] object instance

```
      // assume that in the Drop class we have an instance variable sWidth, sHeight that define
      // the bounding box for our drop's shape
 //assuming the Paddle has x,y,pWidth, pHeight

 //this code is in the Drop Class definition
 //   this is called in the main tab as:   drops[i].isIntersecting(paddle1);

     boolean isIntersecting(Paddle p){
  if(this.y + this.sHeight >= p.y){     //check the bottom point of our drop shape to see if it's hi
          println("y > pY");
          if(((this.x + this.sWidth) >= p.x) && (this.x  <=  ( p.x + p.pWidth))) {
                  println("hit ");
                  this.y=height + 100; // move the drop below the bottom of the visible canvas
                  this.isActive=false;
                  return true;
          }   //end if
     }   //end if
  return false;
}  //end method
```

When we use this intersection method, we'll use it in the main tab, and if the method returns *true*, then we'll want to increment the game score, and set the drop to be inactive. In addition, it's also a good idea to just change the y position of the drop if it's been hit, so that it's off the screen, that prevents display issues.

14.9 Summary

So, in the Paddle class, we have created event handler methods: `pressedLeft()` and `pressedRight()` When we create a Paddle object, paddle1, then we'll have that object call these event handler methods within the global `keyPressed( )` event. The event handler methods are used to trigger object behavior code that we'll need to create within the Paddle class itself, one example of this behavior is the `displayLeftArrow()` method.

Using Object-oriented programming means that we provide more structure to our code. It can be a little confusing to figure out how to organize code when initially learning object-oriented programming. It can be helpful to think about objects as being responsible for knowing how to implement their own behavior. From this perspective, within the main

program, either in the `draw()` or `setup()` functions, we want to tell objects when to implement behavior, either as part of a sequence of functions, or as the result of some event being triggered, but then we want to let the object itself be responsible for knowing how to implement it's own behavior, so that code should be contained within the Class definition.

14.10 Questions:

1. Why have we decided to use `int` as the type for the state variable `direction`?
2. **What is the benefit of creating simple methods like displayLeftArrow( ) which do one** specific task instead of just writing that additional code within the pressedLeft( ) method?

CHAPTER 15

Inheritance

For our game, we want to have a variety of Drop objects. The easiest way to do this is to create new Classes that inherit from the Drop Class. Inheritance represents a hierarchical relationship between object classes, which we can think of as being an *is-a* relationship. In Processing, there can only be 1 level of class inheritance. Other languages allow for deeper object hierarchies. For our example, we'll say that a Seahorse *is-a* Drop, and a Star *is-a* Drop object.

15.1 Polymorphism

One huge benefit of having child class objects is that we can still refer to all of these objects as Drop objects, this is referred to as *polymorphism*. This will allow us to have an array of Drop objects where we can loop through an array of Drop objects, with the actual objects in the array being child objects such as Stars or SeaHorse objects. When we define the Star and SeaHorse classes of objects, we must `extend` the Drop class, the child classes will inherit all instance variables and methods from the Drop class. This will allow us to manage multiple types of dropping objects in the game code, while still referring to these objects as Drop objects.

Let's create 2 child Classes: Star and Seahorse These objects will use the PShape object for their visual display. Let's start with the Seahorse class. We need to use the ‘‘ extends‘‘ keyword to indicate that the SeaHorse class is a child class of the Drop class. As noted above, they will implement PShape for their display, in-fact we'll use an .svg file to create the shape for these objects, so the display() method will need to be implemented in these child classes, so it will over-ride the Drop class display method:

```
//class definition for the SeaHorse class

class SeaHorse extends Drop{
        PShape s;

        SeaHorse(){
                super();
                s=loadShape("seahorse.svg");
        }

display(){
                //code in here to display the .svg file
                println("seaHorse method");
        }

} //end of SeaHorse class
```

15.2 Method Over-ride

So, both the Drop class and the SeaHorse class have code that implements the display() method. So, the compiler must determine which display() method to use if a SeaHorse object calls the display() method. The compiler first checks the child, SeaHorse class, if it has code implemented for a method, when a method has been called by a child object, then the child method code is implemented. Let's clarify this concept of method over-ride. In the main program tab, we'll have a SeaHorse object, and then it will call the display() method. We'll expect that it'll print the text "seaHorse method" to the console since that's the code we've written above in the Seahorse display() method.:

```
//this code is in the main program tab

Drop shorse=new Seahorse();

draw(){
        shorse.display();
}
```

15.3 Arrays of Multiple Types of Objects

An array must be declared to contain a specific type of element. Above we've looked at an array that's been declared to hold Drop elements: `Drop[] drops`. However using the Object concept of Inheritance will allow us to use this `drops` array to hold several different types of Drop objects, as long as these other objects are from a class that is a child class of the Drop class. We

CHAPTER 16

Arrays of Objects

Shiffman's RainDrop Game uses an Array to store Drop objects, so let's look at how to use `Arrays` to store objects. Shiffman's Exercise 10.4 provides the code that we'll review here.

On the processing website, the Array reference provides several examples of how to create an Array and how to initialize the values within an array. It's important to note that Arrays can only contain one type of object, the `type` they are declared with. However, we'll see how Object Inheritance will provide a convenient way to work with different object types in a single array.

Let's create an Array of Drop objects, first we need to declare the Array, then we need to initialize the Array and set the size of the array. Then we need to create a Drop object for each Array element:

```
//Main Program

Drop[] drops;   // here we are declaring the array of Drop objects, with name: drops

void setup(){

        drops= new Drop[10];    // here we are initializing the array and setting the size to 10.

        for(int i=0; i<drops.length; i++){  //initialize each array element
                drops[i]= new Drop( );   //here we are creating a new Drop object for each array ele
        }

}        //end of setup()
```

16.1 Object Cache

In Shiffman's RainDrop Game, he uses the Array of Drops as a way to reuse objects. So, rather than creating a new object each time a RainDrop falls below the bottom edge of the canvas, he identifies it as *finished* so that he can re-use it at a future time. This is common in game programming. So, we need to add a boolean state variable to the Drop class that will let us indicate that a raindrop is not actively showing on the canvas. In addition, we also want to make a method called finished() that set the state of a drop to `finished==true`. This is another example of creating a simple method that does 1 simple task. We'll have another method called `reachedBottom( )` which we can also use to test for inactive drops:

```
//Drop Class Definition
// New variable to keep track of whether drop is still being used

boolean finished = false;
```

```
//Drop Methods
 // If the drop is caught

void finished() {
        finished = true;   //sets the finished state to true so the drop can be reused
}
// Check if it hits the bottom

boolean reachedBottom() {
        if (y > height + r*4) {  // If we go a little beyond the bottom
                return true;
                }
        else {
                return false;
                }
}
```

In the main program these methods, and the boolean `finished` variable are used to control the game. The `finished` variable is used to filter the Drop objects so that the methods are only called on active objects. This type of optimization is common in games. It's much less `expensive` to check the value of a variable than to call a method or function, so the drop methods are only called on active drops. The code below shows that within the main game `draw()` loop, the entire game is based on calling methods of game objects like Drop, Timer, and Catcher. Below is a code snippet where a for loop is used to iterate over the Array of Drop objects and then to call the appropriate Drop methods and increment game variables as needed:

```
      // Move and display all drops
   for (int i = 0; i < totalDrops; i++ ) {
     if (!drops[i].finished) {   //this is a filter so we only process drops which are active
       drops[i].move();
       drops[i].display();
       if (drops[i].reachedBottom()) {
         levelCounter++;
         drops[i].finished();
         // If the drop reaches the bottom a live is lost
         lives--;
         // If lives reach 0 the game is over
         if (lives <= 0) {
           gameOver = true;
         }
       }

       // Everytime you catch a drop, the score goes up
       if (catcher.intersect(drops[i])) {
         drops[i].finished();
         levelCounter++;
         score++;
       }
     }
   }
```

Summary

In this section we have looked at how to use an Array to hold objects and then to allow looping through the array to check an object's instance variables like `finished` which can act as a filter to minimize the number of method calls

that are executed by the program. In addition, we discussed how the `finished` instance variable lets us identify objects that are un-used so we can re-use them at a later time. These types of optimizations are important in game development so the game can execute at a fast speed.

CHAPTER 17

PShape Objects

The Processing website has a good tutorial about PShape objects, so it's advisable that you review that tutorial in order to have a basic understanding of PShape objects. The idea is that PShape allows us to create geometric primitives that we can use as objects. There are a variety of ways to use PShape, let's start by creating a Star class.

17.1 Stars

In the Processing PShape tutorial, the authors create a Star class, and they use the createShape(), beginShape() and endShape() functions, along with a list of vertices in the Star constructor to create the geometry for the Star object.

However, they go on to describe a subtle concept; the idea of creating the star geometry one time in the main setup() function, and then passing that PShape object as an input parameter to the Star constructor. This can cut down on rendering costs, where Processing can essentially 'memorize' the geometry of the single PShape object, rather than having to render the geometry for each one.

For now, let's ignore these suggestions, and we'll create the geometry in the Star constructor function. Below are snippets of code for the Star class, where it inherits from the Drop class. For our current version of the RainDrop game, the Drop class does not inherit from any other class:

```
class Star extends Drop{
        PShape s;

        Star(float _x){  // constructor lets us set the x-position
                super(_x);  // call the Drop constructor as the first line of code

        // First create the shape
                s = createShape();
                s.beginShape();
                // You can set fill and stroke
                s.fill(102);
                s.stroke(255);
                s.strokeWeight(2);
                // Here, we are hardcoding a series of vertices
                s.vertex(0, -50);
                s.vertex(14, -20);
                s.vertex(47, -15);
                s.vertex(23, 7);
                s.vertex(29, 40);
                s.vertex(0, 25);
                s.vertex(-29, 40);
                s.vertex(-23, 7);
                s.vertex(-47, -15);
```

```
                s.vertex(-14, -20);
                s.endShape(CLOSE);
                }
        }
```

The code above is taken directly from the Processing PShape Tutorial, except that we've made the Star class a child class of the Drop class.

Because the Star class inherits from the Drop class, we don't need to explicitly declare most of the instance variables. However, notice that in the first line of code in the constructor that we're calling `super()`, this is the Drop class default constructor.

Shiffman notes in the PShape Tutorial that by default, a PShape geometry is defined relative to the canvas origin (0,0). Therefore, he notes that it's we'll almost always use `pushMatrix()`, `popMatrix()`, and `translate()` in order to locate our PShape objects on the canvas.

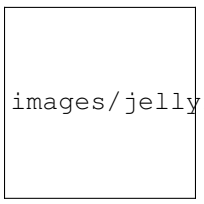
So, to write a `display()` method for our Star, we need to remember to translate the origin of the canvas to the x,y coordinates of our shape before displaying using the *shape()* function. Here's a display method that also lets us set the color of the PShape object:

```
        void display(){
                pushMatrix();
                translate(x,y);     //x,y were given default settings
                s.setFill(color(0,200,127));   //we can change the fill color here if we want
                shape(s,0,0);
                popMatrix();
}
```

If we want to determine where the center of the PShape is, what code can we write? The easiest thing to do is to note that we've translated the origin to the x,y position, so if we create an ellipse at (0,0) then we'll be able to observe where the center is.

17.2 PShape using SVG Image File

Another way that we can use PShape is to load an .svg file. There are may sources for .svg files, for this project, I'm using an svg file from *The Noun Project*. This site has lots of .svg icon files that are free for use if you provide proper attribution for the designer of the .svg file. I'm using an .svg file of a seahorse designed by: Agne Alesiute at http://thenounproject.com/ . I am also using an .svg file of a jellyfish designed by: Eli Ratner. Below is the jellyfish .svg file. We can see that the image has a text component where the designer's name is listed. While we need to provide credit to the designers in our programs, we need to remove that text in order for this .svg file to work in our programs.



17.3 Edit SVG Files

There are 2 ways that we can edit .svg files. Since .svg files are a standardized data format file that list vertices for geometry and font elements, we can open the files in either a vector software system like Adobe Illustrator, or

Inkscape. We can also open the files in any text editor like notePad or textEdit. If we open the file in a text editor, then we need to go to the bottom of the file and find the <text> </text> content section. We need to carefully select this content from the opening tag to the closing tag, and the delete that content, being careful not to delete anything else. In addition, if we look at the .svg file, we can determine the width and height of the object, in addition to other image details. Below is the .svg code for the image above:

```
<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink" version="1.1" id="
<g id="Layer_1_1_">
<g id="Your_Icon">
</g>
</g>
<path fill="#000000" d="M50,8.806c-15.769,0-28.552,10.665-28.552,23.821c0,3.51,1.188,7.938,2.544,9.84
<text x="0.0" y="117.5" font-size="5.0" font-weight="bold" font-family="Helvetica Neue, Helvetica, Ar
</svg>
```

So, delete the <text> </text> line in the code above, however, make sure not to delete the closing </svg> tag. Also, in the first line of code, if the view box is significantly different than the width and height, which are usually 100px by 100px, then this can cause problems. You will probably want to pick a different image.

In order to use an svg file for a PShape object, it's necessary to use the following syntax in order to load the image.

1. The .svg file must be put inside a folder named: `data`, inside your sketch folder
2. PShape s= loadShape("seaHorse2.svg"); // this loads the image
3. shape(s, x, y, width, height) ; //this is used to display the svg.

17.4 SVG Origin

With SVG files, the x,y position refers to the top left corner of the svg file. If you open the svg file in a text editor, you can read the width and height dimensions of the svg. Those can help us when we try to determine the bounding box for collision detection. Below is the display function that is overriding the default style of the svg to allow us to reset the fill color. Below that I've added a rectangle to show the bounding box. Since we're using translate(x,y) so that we've moved the canvas origin to the svg corner point, then we'll draw the rectangle at (0,0).:

```
void display(){
        pushMatrix();
        translate(x,y);
        s.disableStyle();  // Ignore the colors in the SVG
        fill(c);     // Set the SVG fill to myColor
        stroke(55);          // Set the SVG fill to gray
        shape(s,0,0,sWidth,sHeight);
        noFill();
        rect(0,0,sWidth,sHeight);  //bounding box
        popMatrix();
        }
```

17.5 Bounding Box

In order to determine if we have a collision with the paddle, we need to define a bounding box for our PShape object that we'll use to determine contact with the Paddle object. Shiffman's game used circular objects, and determining intersection with circular objects is a bit easier than it is with rectangular objects. As noted above, we've translated the origin to the x,y position of our .svg file, this corresponds to the upper-upper left corner of the shape.

CHAPTER 18

ArrayList

An ArrayList is a *data structure* that allows us to store objects in our programs, and is similar to Arrays in many ways, in-fact, the ArrayList Class uses Arrays, but provides convenience methods to allow us to dynamically resize by adding or removing elements during our program's execution. An ArrayList can be considered a resizable array, we can add and remove elements to an ArrayList using ArrayList methods. An ArrayList provides a good example of how we'll want to build our own classes because it provides access to features like size using a method: size(). That is different than Array because when we use Arrays, we could access the size using dot notation to access the `length` instance variable.

18.1 Declaration

So, we'll create an ArrayList object by calling the ArrayList constructor., however it's slightly different than what we've seen when creating objects because we also need a way to tell the compiler what type of objects we'll store in our ArrayList. For this, we'll use a special syntax `< Type >`, where we'll use angle brackets when we declare and initialize our ArrayList. This may seem odd at first, but you will get used to it and will see it used for other similar types of Classes that hold 'Collections' of Objects.

Here is how we can declare an ArrayList, we need to use a special *type* syntax to specify what *type* of objects we'll use the ArrayList for. This syntax is a *type parameter*.

```
ArrayList<Drop> dropList;
```

For an Array, we would have specified this way:

```
Drop[] dropArray;
```

Notice that we use the angle brackets to specify the type of object `<Drop>`

18.2 Initialization

Since we will normally do initialization separate from the declaration, below is the syntax for initializing an ArrayList compared to how to initialize an Array.:

```
ArrayList<Drop> dropList;  //declare
dropList = new ArrayList<Drop>();   //we're calling the ArrayList constructor
```

Below is the Array syntax we've been using where we have to declare the size of the Array when we initialize it, the Array syntax is quite different from other object constructors we've worked with, it has it's own unique syntax whereas the ArrayList is similar to other objects.:

```
Drop[] dropArray;       // declare
dropArray = new dropArray[ 6 ];   //initialize
```

18.3 Add an element

Below we'll add an element to an array, we use the array name and bracket with index value to initialize an element:

```
``dropArray[1]= new Drop();     //standard Array notation``
```

For an ArrayList we use the `add()` method of the ArrayList object, and our new element is added to the leftmost end of the ArrayList. In the example below we are creating a new Drop element that has not been assigned to a reference variable. This is a convenient way to create and add objects to an ArrayList:

```
dropList.add(new Drop());  //ArrayList add( Object e ) method takes an object as the input parameter
```

Or, if we already have an element, we can just add that by passing the reference variable as the input argument to the add() method as below:

```
Drop drop1 = new Drop();

dropList.add(drop1);
```

18.4 Access Elements

One common method to access and use an element in the ArrayList is to use a `for loop` and an iterator variable, `i` , and loop through the ArrayList using the ArrayList size() method to control the for loop. If you are not making any modifications such as adding or removing elements, a normal for loop syntax works fine as shown below, however this syntax won't work if you're adding or removing elements in the array.:

```
for(int i=0; i< dropList.size() ; i++){
                Drop d= dropList.get(i);   //you must create a Drop reference variable to retrieve a
                d.display();  //do something with the current drop
}
```

18.5 Remove Elements - Caution when Looping

In order to use a for loop to access ArrayList elements when we know we'll be removing some elements, we need to remember that the ArrayList is actually using an Array, so we need to be careful to stay within the bounds of the array, and to make sure we're able to access each element as we expect.

Let's look at a simple example. If we create an ArrayList with 2 elements, then they will have index values of 0 and 1. If we were to use the loop above, then i would start at 0, we'd use get(i) to retrieve the first element.:

```
//if dropList has 2 elements,

for(int i=0; i< dropList.size() ; i++){     //incorrect syntax, this causes problems
                dropList.remove(i);
}
```

The first time through, i=0, we remove the first element from the array. Then, we increment i, so i=1. We still check to make sure that i < droplist.size(), and now the size() is 1. So, we don't enter the for loop, and we haven't actually examined the remaining element in the list. Let's look at another way to do this.

18.5.1 Reverse For Loops

A simple syntax change so that we go through the loop in reverse will prevent the problems from above. So, we need to restructure the for loop: We'll start at the largest value: droplist.size(), then we'll stop once we've reached the first element and it has an index of 0. Each time, we'll subtract 1 from i. If we have a list of 2 elements, size() = 2 and their indexes are 0,1. So we want to make sure to subtract 1 from the starting value so our starting value is `droplist.size( ) -1` The for-loop structure is: (initializer ; test-condition; iterator)

```
for( int i= droplist.size() -1; i > 0 ; i--){     //correct syntax, this works
                droplist.remove(i);
}
```

So, the first time through, i=2-1, so i= 1. So we'll remove the last element in the list, so the list now has 1 element in position 0. The value of i is updated by subtracting 1: i=1-1, so i=0. So, now when we enter the loop, we're looking at the element at position '0'. So we've at least looked at each of the elements, the code above removes this as well, but the take-away is that a reverse for-loop allows us to make sure we iterate through each element in the list.

So, if we need to change the code in our game so that we're using ArrayLists instead of Arrays, then we can eliminate some of the book-keeping variables and simplify the code.

18.6 Paddle Drop Game using ArrayList

In the Paddle Drop Game, there are 2 main sections where we need to change the code if we want to implement using ArrayLists instead of Arrays:

1. When we create each new Drop element as the timer isFinished()
2. When we loop through to move each drop, check for collision, check for the drop being finished

18.7 Creating Drop in drops ArrayList

For the part where we create new drops, we still want to use random( ) so that we randomize the type of child drops created at each step. We had used the variable totalDrops to limit the total number of drops created for a level, we actually still need to do that in order to know when to increase the game level. Below is the code where we used Arrays to hold our Drop objects in the game:

```
  if(timer.isFinished()){
  if(game1.totalDrops<drops.length){
      int choice = (int)random(0,2);  //  gives 0,1 vaules
      //println("choice " + choice);
      switch(choice){
           case 0:  drops[game1.totalDrops]= new Seahorse();   //change this to ArrayList
               break;
           case 1:  drops[game1.totalDrops] = new Drop();     //change this to ArrayList
               break;
         }    //end switch
        game1.totalDrops++;
      }
        timer.start();
}  //end if
```

We'll now change it to use a drops ArrayList:

```
    if(game1.gameState==game1.ACTIVE){ // put most active game code in here
if(drops.size() < game1.totalDrops)// totalDrops is the max # of drops per level, we never modify it
    if(timer.isFinished()){
        int choice = (int)random(0,2);  //  gives 0,1 vaules
         switch(choice){
             case 0:  drops.add( new Seahorse());
                 break;
             case 1:  drops.add(new Drop());
                 break;
           }    //end switch
        timer.start();
      }  //end if
```

In the above code section, we're using game1.totalDrops in a slightly different way, now it is as the maximum number of drops that we'll let the game create. We never increment that value, but we'll use it again to check whether we need to advance the game to the next level.

18.8 Iterating through drops ArrayList

Now let's look at how we'll access a single drop and process each of the drop steps, first let's look at how the code was written when we used arrays:

```
///loop through array and deal with each drop[i]
       for(int i=0;i<game1.totalDrops;i++){
              if(drops[i].isActive==true){  //only look at active drops
                  boolean isHit=false;  //reset each time through for loop
                  drops[i].move();
                  isHit= drops[i].isIntersecting(paddle1);   //check to see if intersect with paddle
                      if(isHit){
                              //println("isHit");
                              drops[i].caught();
                              game1.score++;
                              game1.levelCounter++;  //count caught drop

                          }
                 if(drops[i].reachedBottom()){  // reached bottom without getting hit, so lose lives
                   drops[i].finished();   //set drop to be not active

                   game1.dropFinished();
                   game1.levelCounter++;  //count each drop that is no longer active
                   game1.lives--;  //life is lost
                   if(game1.lives<=0){
                     game1.gameState=game1.END;   //set the game to over
                   }
                 }
              drops[i].display();
              }  //end isActive
      }//end drop[] loop
```

Now we need to modify this code so that we're accessing elements from the drops ArrayList. As mentioned above, we need to modify the for-loop so that we're going in reverse order, using the syntax that section above. Instead of using totalDrops as the maximum value, we need to use the current size of the drops ArrayList. So we'd start with rewriting the for-loop and then we need to select a single Drop object to process, instead of using drop[i], we need to create a Drop reference variable to act as a pointer for each element we retrieve from the ArrayList:

```
  for(int i=(drops.size()-1 ); i > 0 ; i--){
        Drop d = drops.get(i);

  //change all of the following code so we're using d instead of drops[i]

  // REMEMBER:  we should remove a drop if it's been hit or is finished, we need to use the current
  //index

        drops.remove(i);

}
```

Then we also need to look at the rest of the code and determine if there are areas where we are using the drops Array, and we need to fix that code. In Shiffman's code, he used the variable `levelCounter` as a way to keep track of how many drops have been removed from the game, either due to being hit or to falling off the screen. We need to re-examine this logic. Currently we have this test to determine if we should increment the level: if(game1.levelCounter >= drops.length). As a first step let's compare levelCounter to totalDrops.

Goal: Have logic to control changing levels:

1. levelCounter and totalDrops are now part of the game class, they control how many drops are created for each level
2. totalDrops is set to some max value like 30, and never changes. //Infact, we should consider this a constant and capitalize it!
3. levelCounter is incremented each time a drop is removed from the screen, either isHit or isFinished
4. drops.length was an Array instance value, it gave the fixed size of the array.
5. drops.size() is an ArrayList method that gives the current size of the ArrayList
6. **We need to compare a changing value (levelCounter), with a fixed value: totalDrops.**
   Otherwise our game will continue to make and remove drops from the game forever.

Here is the old code version:

```
          // If all the drops are done, that level is over!

if (game1.levelCounter >= drops.length) {
    // Go up a level
    if(game1.levelIndex<levels.length-1){
        game1.levelIndex++;
        currentLevel=levels[game1.levelIndex];
        paddle1.pWidth=currentLevel.paddleWidth;
    }
    // Reset all game elements
    game1.levelCounter = 0;
    game1.lives = 10;
    game1.totalDrops = 0;
    timer.setTime(constrain(300-game1.levelIndex*25,0,300));
     }
```

In the new code version, we'll replace that if-condition with:

```
//set totalDrops to some constant value like 30
if(game1.levelCounter >= game1.totalDrops){

        //Can we increase the level?
        if(game1.levelIndex<levels.length-1){
      game1.levelIndex++;
```

```
      currentLevel=levels[game1.levelIndex];
      paddle1.pWidth=currentLevel.paddleWidth;
  }

                //Reset all game elements We should put this in a game method: resetLevel( )

        game1.levelCounter = 0;
        game1.lives = 10;

        // We don't want to reset totalDrops to 0!
        // game1.totalDrops = 0;   //Don't reset totalDrops

        // clear the arraylist of any remaining elements
        drops.clear();

        timer.setTime(constrain(300-game1.levelIndex*25,0,300));

} // end if
```

This should provide a good start for how we can change our game so that we're using ArrayLists to hold our drops instead of using Arrays.

Make sure that you understand that you need to modify the game logic when you switch to using ArrayLists instead of Arrays, the logic should be simplified.

CHAPTER 19

Abstract Classes

An Abstract class cannot be used to create objects. Instead, Abstract classes function to provide a base class for sub-classes to inherit; the idea is that we can define a generic class that has common instance variables and methods that will be shared by all sub-classes so they are defined in this abstract base-class.

The generic idea of a Dropping object in a game inspired us to create the Drop class, and we also decided that we'd like to have a variety of specialized Drop objects, like SeaHorse so we used sub-classes to make these specialized versions of the generic Drop class.

In the games we are designing, we have used the Drop class as a base class for sub-classes like SeaHorse and Star. We can use the powerful structure of polymorphism to allow us to refer to all sub-class objects, such as SeaHorse objects, as if they were actually Drop objects. This has allowed us to create Arrays and Array Lists of Drop objects, and we've been able to put SeaHorse, Star, and Drop objects in these containers.

We can decide that we only want to allow sub-class objects in our game, since we've created each of them to use distinctive .SVG icons to enhances our game's aesthetic appeal.

In our code, to make the Drop class abstract is very simple, we just include the abstract keyword before the class definition. Once we add this `abstract` keyword to a class, we can no longer make objects directly from that class.:

```
abstract class Drop{
                ///class definition code
}
```

19.1 No Abstract Objects

Once we've made this decision to never create 'plain old' Drop objects, do we still even need the Drop class? The answer is YES! The Drop class is helpful because it allows us to define a common set of properties or instance variables, and a common set of behaviors, called methods. In addition, we can also require that all child classes provide their own implementation of certain methods, like `display()`, which we have decided are essential Drop object behaviors, yet they are distinct for each child object. These required method over-rides are called 'abstract methods'.

19.2 Abstract Methods

Abstract Methods are methods that we require to be implemented within each inheriting sub-class. For our programs, `display()` is a method that would probably be abstract because we know that each different type of child object will have some unique type of object display. For some sub-class objects, their display may include some type of special animation like rotation or scaling etc.

When we define a method as `abstract`, we don't include any part of the method body in the base-class, we only include the method signature, so we know what return type and parameters that we must implement in sub-classes. In addition, the compiler will give us an error if we forget to implement that method in a sub-class. However, it's important to note that the abstract class can still have methods like `move()` that are not abstract. These methods have code within the curly braces that is executed when the method is called. A method like `move()`, which has the same code for all sub-classes, should not be abstract, and the method should be implemented in the base-class so it can be used by all sub-classes. Below is the syntax for creating an abstract method in the base-class:

```
abstract class Drop{
        // instance variables
        // constructor

        abstract void display();   //abstract method declaration syntax

        void move(){   //move is not an abstract method
                y += speed;
        }

}
```

19.3 Abstract Method - Sub-class Implementation

The `display()` method was declared as `abstract` in the Drop base-class. So any class which extends Drop must now implement `display()` or there will be a compiler error. In the SeaHorse class, we have already written code to implement this method, so no changes are required.

Summary: This added *abstract* class structure might seem like a lot of extra work for no benefit, however these added structures makes our programs easier to extend and easier to debug.

CHAPTER 20

Interface

An interface defines a set of behaviors that are meaningful across several different types object classes. Because a class can only have a single parent or base class, interfaces provide another method to define organizational structures for our programs. Whereas a child-class can *extend* a base-class through *inheritance*, a class *implements* an *interface.* An interface is similar to an *abstract class* because it is defining methods that must be implemented in class that implements the interface.

For our game, we might want to have some of our game drops explode if they hit our paddle. However, if we don't want all of the drops to explode, how can we take advantage of polymorphism, where we have an ArrayList of Drop objects, where some of these objects can explode and some don't explode? We have defined the Drop class to be an `abstract class`, where the display() method is an abstract method that must be implemented by all child classes.

For the example code below, we have 2 sub-classes, CircleDrop and SquareDrops, where we only want the CircleDrop objectsto be Explodable. To do this, we'll have it implement the explode() method from the Explodable interface.

To use an interface, first we must define the interface, this is similar to the syntax for defining a class. Since an interface is similar to an abstract class, we will only provide method signatures, and no method body code for the methods that we are defining for the class. The methods are abstract by default, so the abstract keyword is not required, however only the method signature can be part of the interface definition. The second thing we need is a class that will implement the interface, and that class must provide method definition code for each method defined in the interface that it is implementing.

20.1 Part 1: Define the interface

Interface Definition:

```
//an interface is not a class, but has a similar syntax, notice the keyword: interface

interface Explodable{

void explode();  //  method signature only

float getExDimension();  // method signature only, provides access to child class attribute
                                                    //this is the dimension that we'll expand for explos
}
```

20.2 Part 2: Some Class implements the interface

Interface Implementation:

```
// class circleDrop implements the Explodable interface

    class CircleDrop implements Explodable{
            float x,y,radius;

            CircleDrop(){    //default constructor
                    x=random(0,width);
                    y=random(0,height);
                    r=10;
            }

            void explode(){    //this Explodable method must be defined in this class
                r = r + 5;
            }

            float getExDimension(){ //this Explodable method must be defined in this class
                    return r;
            }

    }
```

20.3 Part 3: Combine Inheritance and Interface

It is important to note that a child class can use both inheritance and multiple interfaces using the keyword syntax below:

- Inheritance: extends
- Interface: implements

The use of Inheritance and Interfaces can be combined, for our project this allows us to create a child class that extends Drop, and implements Explodable:

```
class CircleDrop extends Drop implements Explodable{

            float radius;

            circleDrop(){
                    super();    //call the Drop constructor
                    radius=10;
            }

            void display(){  // this Drop abstract method must be defined in this class
                    ellipse(x,y,r*2,r*2);
                    }

                    void explode(){
                            r = r + 5;
                    }

                    float getExDimension(){ //this Explodable method must be defined in this class
                    return r;
                    }

}
```

20.4 Part 4: Polymorphism and Interfaces:

For our game, we have used polymorphism to allow us to create a Drop Array which contained child objects. We can also use polymorphism with interfaces, similar to how we have used the abstract Drop class, once we made Drop an abstract class we could no longer create Drop objects. Similarly, while we never create an Explodable object, we can create a reference to an object that implements Explodable, and we can check whether each Drop object instance is an Explodable type of object. The use of abstraction through polymorphism means we can have many different types of drop objects, where only some of these are explodable, and we can operate on all of these Explodable using an Explodable reference.

20.5 Instanceof and TypeCast

We can use the `instanceof` keyword to determine if a dropList instance is an object that implements Explodable. Here we have an ArrayList: droplist of Drop sub-class objects, we are iterating through the list in reverse order in case we want to remove an element from the list

- Check to see `if(d instanceof Explodable)`
- Create an reference variable of type `Explodable e`
- Use *typeCasting* to convert `d` to an object that can call `explode()`

Below is the code for this:

```
    for (int i = dropList.size()-1; i >= 0; i--) {

    Drop d=dropList.get(i);
//test to see if the object instance implements the explodable interface

if(d instanceof Explodable){
   Explodable e=(Explodable)d;   //type cast
   e.explode();   //call explode() on the objects that implement Explodable

   if((e.getExDimension())>80.0){  //check size limit
       dropList.remove(i);
   }
 }
```

Below is a link to the example program.

https://utdallas.box.com/InterfaceExampleZip

CHAPTER 21

Simple Audio

The Minim library in processing can be used to add sound to programs. Look in the processing examples from the edit menu. Then go to the Minim Library and select the example: PlayAFile. The code in this project will play a .mp3 audio fole that is contained in the data folder in your sketch folder.

Below is an extremely simple code example demonstrating how to create a sound based on whether the user's mouse is pressed and is on the red ellipse.:

```
//Main project code
///you must include the sound file in a data folder inside your sketch folder

//import the Mimin library
import ddf.minim.*;

//declare minim object and AudiPlayer objects

Minim minim;
AudioPlayer sound_hit;


void setup(){
  size(300,300);
  //you must call the Minim constructor so it can find the audio file in the data folder
   minim=new Minim(this);  //pass a reference to the current 'sketch' to the constructor
   sound_hit= minim.loadFile("missile.mp3");
}

void draw(){
  background(255);
  fill(255,0,0);
  ellipse(width/2,height/2,40,40);
  if(mousePressed && mouseX < (width/2+20) && mouseX >(width/2-20) && mouseY <(height/2+20) && mouseY
                ///rewind and play each time through the draw loop if mousePressed is true
                sound_hit.rewind();
                sound_hit.play();
        }

}
```

Here is a link to a zip file of the code so you can see how to create a data folder to hold your mp3 file.

https://utdallas.box.com/simpleAudio

Mimim Library Reference: http://code.compartmental.net/minim/

For Minim, you will use:

- loadFile()

For the AudioPlayer, you might use:

- loop()
- isLooping()
- play()
- isPlaying()
- pause()
- rewind()

There are lots of other methods, these are just a small sample, but they are the ones you'll probably end up using.

CHAPTER 22

Game Programming

Learning to design and program simple games provides a good context for learning many fundamental computing concepts such as user interaction, collision detection, conditional branching, boolean logic, object oriented design, and state machines.

22.1 Links to Game Programming Tutorials

Listed below are links to some excellent online tutorials and blogs that focus on simple game programming using the Processing language.

22.1.1 Dark Views Writing Games with Processing.

In this set of blog posts, the author creates a set of tutorials about how to develop games in processing. He uses an iterative approach to design a simple game. He uses an object oriented approach to create a simple game and includes details on how he'd refactor his code after his initial prototype.

22.1.2 Dr Doane Thinking Through A Basic Pong Game in Processing.

This blog provides a very good overview of the problem-solving approach to designing a basic Pong game using Processing. This tutorial uses functional decomposition rather than an object oriented approach so this tutorial can help students understand basic function design including basic collision detection.

22.1.3 Cate Huston Let's Make a Simplified Game of Pacman.

This blog tutorial creates a very simple version of Pacman.

CHAPTER 23

Resources

23.1 Books

Learning Processing by Daniel Shiffman

> The Learning Processing website provides supplementary information to support the textbook including example code and answers to exercises.

The Nature of Code by Daniel Shiffman

> The Nature of Code website provides full access to the entire contents of this book and includes interactive code for most code examples.

23.2 Websites

Processing.org

> The Processing website provides code examples, tutorials, reference documentation and other supplementary resources.

Learning Processing

> Website to accompany Daniel Shiffman's book which we'll use as the course textbook.

Open Processing

> The Open Processing website provides an online code editor and cloud based storage to allow sharing of code examples. There are many searchable code examples which are interactive using either the java applet or they are embedded using processing.js. This website is designed to support and sustain the processing community.

CHAPTER 24

Glossary

24.1 Abstract Class

An abstract class is a class that cannot be used to instantiate objects, it is designed specifically to be used as a base-class for sub-class inheritance. Abstract classes provide structure: instance variables and methods, that sub-classes will inherit. Abstract classes must be extended by base classes, they are explicitly designed to function as a base-class.

In addition, abstract classes can have abstract methods, these methods define a contract that child classes must enforce, any child class that extends an abstract class must implement code for any abstract methods defined in that class. Abstract methods allow us to use polymorphism, for our game, we have defined Drop as an abstract class. This means that we can never create Drop instances. However, we can create Drop-type references, we can think of these as pointers that are of type: Drop. These reference-pointers can point to any object instances that are from a child class of the Drop class, in our game, these would be starfish, stars, jellyfish, etc. The benefit of having Drop as an abstract class is that we can define all common instance variables within the Drop class, and we can also define any common methods, where all sub-classes use the base-class method. move( ) is an example of a method that is the same for all sub-classes, so we can define it one time, in the base-class: Drop. This prevents duplicated code, improves our program's organization, and makes it easier to extend our program by simplifying the process of adding new sub-class objects.

24.2 Abstract Methods:

Abstract methods are used in abstract classes and interfaces. These methods define a contract that child classes must enforce, any child class that extends an abstract class must implement code for any abstract methods defined in that class. Similarly, any class that implements an interface must provide code to define the methods that are declared in the interface. Method declaration consists of the signature of a method, this includes the return type, the method name, and the input parameters for the method. Abstract methods are methods that have unique meaning for each class, such that it makes sense for them to have their detail specified at the sub-class level. The use of abstract methods provides a means to define a structure and require that it's enforced by all classes that implement or extend that defined parent-class or interface construct. In contract, there are many methods that will be identical across sub-classes, these should not be abstract and should be defined at the base-class level. There is a defined syntax to specify non-abstract methods for interfaces, however this is not supported in Processing, and all interface methods are abstract by default in Processing.

24.3 Interface:

An interface is very similar to an abstract class. Similar to abstract classes, interfaces can't be used to create object instances. In fact, by default, methods of an interface are abstract. This means that there's only method declaration

in an interface. Typically, interfaces don't have variables, however constant variables can be defined for use in an interface. A class can `implement` an interface, whereas, a class can `extend` other classes when using inheritance. The important feature of interfaces is that a class can implement a unlimited number of interfaces, whereas a class can only extend 1 base-class in Processing. Therefore, interfaces are used to provide an abstract structure for common concepts such as *comparable*, *clickable*. For our game we developed an *Explodable* interface, and determined that only some of the Drop child classes will implement this interface. The *Explodable* interface has an `explode()` method that must be defined in any class that implements *Explodable*.

24.4 Function Overloading

Function overloading occurs when there are several different variations of a single function. Each version of a function must have a unique function signature, specifically, the arguments for a function must be unique between function versions, the compiler must be able to determine which version of the function is being called, the compiler can distinguish between versions based on the argument types used when a function is executed. A common example of this in Processing is seen with the color-type functions like `fill()`. There are several versions of the fill() function; if there is only 1 input parameter such as: fill(float grayVal), then the function interprets that to set the grayscale value. If there are 2 input parameters, then the second argument is used to set the alpha value. fill(float grayVal, float alphaVal). Function overloading is a powerful way to implement slightly different behaviors depending on the number and type of input parameters for a function. Another example of overloading occurs with class constructors, a default constructor takes no input parameters, while other constructors for an object take input parameters to set some default values for initialization of instance variables.

24.5 Method Overriding

Method overriding is a feature of Inheritance in many Object-Oriented languages; where both the parent and child class implement identical versions of a specific method. In the Processing language, many of our custom classes have implemented a '' display()'' method, since many of our programs create visual designs, animations, or games. The parent class and child class both implement displqy() methods that have identical method signatures, so the compiler follows a specific protocol to determine which method should be executed when a child object calls a method. When a parent class object executes a method, then the parent class's method code is executed. However, when a child class executes a method, the compiler first looks at the child class definition to see if the method has been implemented within the child class definition, if it has been implemented in the child class, then that child version of the class method is executed.

24.6 Object Inheritance

Inheritance is an important feature in most Object-Oriented languages. Inheritance in programming reflects the idea that the concepts that we're trying to represent in our code often have natural hierarchical relationships, where objects can be categorized as belonging to a broader type of object. These object hierarchies can be useful for organizing our program code and is one of the main benefits of Object-Oriented languages. We easily understand that *animal* is a broad category of creatures, where there are many different types of animals such as dogs and cats. We are comfortable with the concept of Animals being a broader category things than the categories of dogs or cats. If we were to create a program about an animal shelter, it would be nice for us to be able to make use of these relationships between these types of objects in our program. We can do this by using the specific syntax keyword `extends` when we define the Dog and Cat classes, where we would need to first define the Animal class. The benefit of using inheritance in our programs is that we can eliminate duplicate code, and we can also use polymorphism as defined below to allow the use of the base class as a way to reference sub-class objects.

24.7 Polymorphism

Polymorphism is one of the main benefits of using the Object-Oriented concept of Inheritance, where a child class is designed as an extension of the base or parent class. Polymorphism means 'many-shaped', and in Object- Oriented programming, it means that we can create a child object, using a reference variable of the parent base class. In our paddle game, we created an array of Drop objects, where *polymoprhism* allows us to have Star and SeaHorse objects contained within the array of Drop objects. For a single element, the code below uses a Drop object reference which we then assign a SeaHorse object, then we can re-assign a Star object. It is important to note that this is a directional, or one-way relationship, we can't use a reference from a child class to point to a base or parent class object as shown in the last line of code below:

```
Drop drop1;      //Drop object reference can point to Drop objects and objects which extend the Drop
Star star1;
drop1 = new SeaHorse();    //SeaHorse is a child class of the Drop class
drop1 = new Star();                    //Star is a child class of the Drop class

star1= new Drop();    ///ERROR
```

24.8 Static Variables

Processing does not support traditional static variables, the only way to have static variables is to have the entire class declared as a static class. We aren't going to cover Static methods, classes or variables in this class, however this is an important concept that will be covered in the next course.