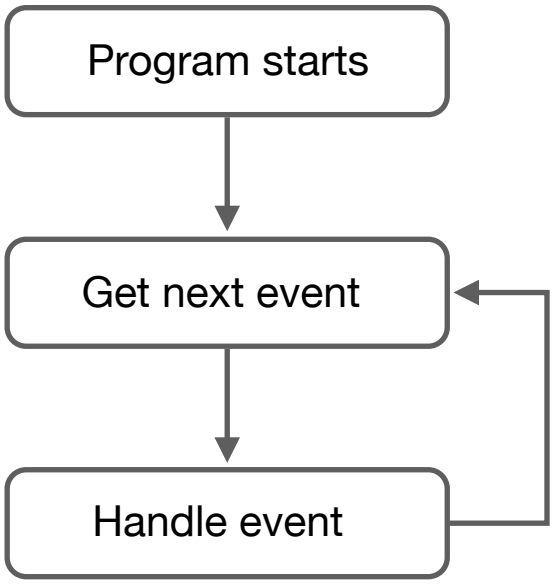
Interaction

Event-loop program

Most of the programs we will create this semester will be interactive. So far, we have only talked about instructions working in a linear sequence - start at the top, and perform the instructions until you get to the bottom. How does the sequence of instructions work for an interactive program? Logically, you can consider an interactive program to be organized like the following diagram.



The program starts, and when it begins there may be some initialization that needs to be done to prepare the program for its job. Then it enters the interactive portion of its lifetime. The program will wait for an event to occur. There are many possible events that can happen - including keystrokes on the keyboard, mouse clicks, and many more. When an event happens, some set of instructions will be run to respond to (or handle) that event. This pattern is sometimes called an event-loop program, or sometimes an event driven program.

Swift / Tin framework scene update pattern

Our Swift programs using the Tin framework will follow this pattern. In the project there is a Scene object which takes responsibility for managing the visual display of the view. (A view is a specific term for macOS programmers. It means an area of an application's window where drawing can take place.) The Scene object has a function named

update. For now, all we need to know is that a function is a set of instructions. Update's purpose is to perform any drawing needed for the view.

```
override func update() {

  // drawing instructions go here

}
```

We want a dynamic view, which changes over time. To create that, the Tin framework will create an "update event" that repeats 60 times per second. Whenever the update event happens, the update function is called upon to perform its instructions. We chose 60 times per second as the default update rate because our computer display is refreshed at that rate.

The Scene object can also - optionally - provide another function named setup. The setup function instructions are performed once, just before the first update event. This is useful in some cases, when you want to perform one-time initialization before drawing is performed.

```
override func setup() {

  // setup instructions here

}
```

Pay Attention to Blocks of Code

As you look at Swift code, pay close attention to the structure - to the various "blocks" of code. A block of code is delimited by a pair of curly brackets.

```
{
  A block of code
  {
    A block inside a block of code
  }
}
```

These small curly bracket (or brace) characters { and } are very important. You wouldn't ignore a period (.) or comma (,) in your English writing class - would you? No, you would not. You must ALWAYS pay careful attention to any curly brackets in your code. Do not misplace or accidentally delete these bracket characters, they are essential.

Notice that curly brackets are always paired. Every opening curly bracket has a corresponding closing curly bracket. A common early programming error is neglecting these curly brackets. Don't forget to add closing brackets. Be careful when cut, copy and pasting that you include both open and close brackets. Again, remember they always are pairs.

A standard convention is to indent the code inside a block by one tab stop. The indentation is not significant to the Swift compiler. However, using indentation is a very useful technique for improving the readability of a program.

Mouse Pointer

To create interaction we need more than a dynamic display, we also want user input. The first input we will work with is pointer location inside the view. The Tin framework updates a set of useful variables for us to use, and the current mouse pointer location is available. These variables are available through the "tin" object. (We will discuss objects in more detail later.)

```
tin.mouseX  // x-axis location
tin.mouseY  // y-axis location
```

Each time the update function is called to refresh the view, these two variables will have been updated with the current pointer location. These variables can be used like a variable that you declared yourself. For example, with the following instruction a circle is drawn that will be located at the mouse pointer location.

```
ellipse(centerX: tin.mouseX, centerY: tin.mouseY, width:30.0, height: 30.0)
```

A useful possible input could be, set a value or perform a computation based on the relative position of the mouse pointer in the view. You can set that up by using the width of the view so that you will have a number that will be 0.0 when the mouse is at the left edge of the view, 1.0 when the mouse is at the right edge of the view, and values in between 0.0 and 1.0 when it is in between. When the mouse is at the center, it will be 0.5. Using this 0 - 1 range is easy to work with.

This uses another variable named tin.width which stores the horizontal size of the view.

```
var horizontalValue = tin.mouseX / tin.width
```