An Introduction to Object-oriented Programming in MATLAB

Enrique P. Blair, Ph.D.

April 20, 2021

Contents

1 Introduction to Object-oriented Programming in MATLAB

1.1 Object-oriented Programming

Object-oriented programming (OOP) is a powerful approach to writing complex software suites that handle non-trivial tasks and calculations. In OOP, the programmer defines **classes**, which are user-defined, composite data types with associated functions known as **methods**. methods define how an end-user interacts with **objects**, which are **instances** of a class. Thus, programming which involves the use of classes is called object-oriented programming (OOP).

1.2 Data Types

1.2.1 Elementary Data Types

In MATLAB, basic or "elementary" data types include `double` (double-precision floating-point data) and `char` (characters). Numbers in MATLAB are stored in memory as data of type `double`. MATLAB also handles arrays of `double`s, which also are said to be of type `double`.

Additionally, MATLAB has variables of type `char`, which store lowercase letters (`'a'-'z'`), upper-case letters (`'A'-'Z'`), numerals (`'0'-'9'`), and myriad other special characters. **Strings** are arrays (typically, row vectors) of `char`s, and are themselves considered to be of type `char`.

1.2.2 Arrays

Arrays store multiple values of the same data type. Each individual value in an array is called an **element**. In MATLAB, arrays are formed by concatenating elements within brackets (`[` and `]`). Horizontal concatenation is achieved by grouping elements within brackets but delimiting (separating) them either by commas (`,`) or by white space. Vertical concatenation is achieved by delimiting constituents by a semicolon (`;`). Examples of this are given in the command-line sample below:

```
1   >> A = [1 2 3]; B = [4 5 6]; C = [A B]
2
3  C =
4
5       1     2     3     4     5     6
6
7  >> D = [A; B]
8
9  D =
10
11      1     2     3
12      4     5     6
```

Listing 1: The Command Window input and output demonstrates horizontal concatenation and vertical concatenation. Concatenation is accomplished using brackets `[` and `]`. Elements of a horizontal concatenation are separated by whites space or commas (`,`); and elements of a vertical concatenation are specified using a semicolon (`;`).

In line 1 of Listing 1, we define `A` as the horizontal concatenation of 1, 2, and 3; and `B` as the horizontal concatenation of 4, 5, and 6. Then, `C` is formed by horizontally concatenating `A` and `B`. The output of line 1 is shown in lines 3-5. Finally, in line 7, `D` is formed by vertically concatenating

A and B, with output on lines 9-12. Similar concatenation may be achieved using strings. Similar concatenations may be done using data of type `char`. To form an array, all elements have the same data type and the same size in memory.

MATLAB also has another type, known as a `cell` (short for “cell array”), in which each element may be of a different type or of different sizes in memory. MATLAB `double` arrays of different sizes, `char` arrays different sizes, and even other `cell`s may be elements of the same `cell`.

1.2.3 Structures

MATLAB allows the formation of **data structures** (or simply “structures” for short; and `struct`s in MATLAB). A structure is a composite variable comprised of multiple pieces of data. MATLAB `struct`s are very flexible, as a `struct` may contain data of elementary data types, arrays, cell arrays, other `struct`s, or even more complex and elaborate data types (classes). The constituent data associated with a `struct` are called **fields**. We use “dot syntax” to define, assign, and reference the fields of a `struct`. For example, the following input (line 1) defines a new `struct`, `S`, with one field, `a`, and the `double` 7 is stored in the field `S.a`. Then, in line 9, a new field, `b` is added to `S`, and `char` array `'life'` is stored in `S.b`.

```
1  >> S.a = 7
2
3  S =
4
5    struct with fields:
6
7      a: 7
8
9  >> S.b = 'life'
10
11 S =
12
13   struct with fields:
14
15     a: 7
16     b: 'life'
17 >> y = S.a + 5
18
19 y =
20
21     12
22
23 >> disp(['The word is: ', S.b])
24 The word is: life
```

Lines 17 and 23 show how the values stored in `S.a` and `S.b` may be referenced and used in further calculations.

Additionally, a `struct` may be formed by specifying field-value pairs in the `struct()` command:

```
>> R = struct('a', 7, 'b', 'life')

```

```
R =

  struct with fields:

    a: 7
    b: 'life'

>>
```

structs are particularly useful for storing various pieces of information associated with one system or event in real life. For example, conditions (location, temperature, pressure, time, etc.) and data measurements for a particular experiment may be stored in the same `struct`. Or, you might want a `struct` to store the name, phone number, e-mail address, website and other information for a particular friend, relative, or contact. Then, a collection of such `structs` can constitute an address book.

1.3 Functions

Functions are an important element of OOP, so we discuss them here only briefly. In contrast, students in computer-related disciplines may take an entire course on functional programming.

Functions enable code to be modular. A well-written function does a specific task or returns an output based on a set of inputs. The details of the function may be transparent to the user, who can repeatedly call a function without repeatedly copying and pasting the code that underlies the function. This makes the user's code clearer and more understandable. Additionally, if the function requires modification, the implementation of the function may be modified without changing the user's interface of the function.

1.4 Pre-defined Functions

MATLAB has numerous pre-defined functions. Here, we discuss a few functions that will be used in following sections of this document.

1.4.1 Mathematical Functions

MATLAB has numerous mathematical functions for performing calculations, including `sin()`, `cos()`, `real()`, `imag()`, `sinc()`, `exp()`, and so many more.

1.4.2 The `class()` Function

This returns the data type of a variable. In the following code, we define a few variables of different types, and then we use the `class` function to identify their type.

```
>> a = 5;
>> class(a)

ans =

    'double'

>> class(pi)
```

```

ans =

    'double'

>> b = rand(3, 4);
>> class(b)

ans =

    'double'

>> class('s')

ans =

    'char'

>> class('cat')

ans =

    'char'

>> friend1.name = 'Alice'; friend1.age = 23; class(friend1)

ans =

    'struct'
```

Here, we see that numbers and arrays of numbers are of class (type) `double`, and characters and arrays of characters are of class `char`.

1.4.3 The `length()` Function

The command `length(X)` returns the number of elements in the array `X`. Some examples of this are shown in the following listing:

```
>> A = 1:7

A =

     1     2     3     4     5     6     7

>> length(A)

ans =

     7
```

```

>> B = (12:16)'

B =

    12
    13
    14
    15
    16

>> length(B)

ans =

     5
```

For a matrix `X`, the command `length(X)` returns the size of the largest dimension of `X`:

```
>> C = rand(3,4)

C =

    0.6541    0.4505    0.9133    0.5383
    0.6892    0.0838    0.1524    0.9961
    0.7482    0.2290    0.8258    0.0782

>> length(C)

ans =

     4

>> D = rand(4,3)

D =

    0.4427    0.7749    0.3998
    0.1067    0.8173    0.2599
    0.9619    0.8687    0.8001
    0.0046    0.0844    0.4314

>> length(D)

ans =

     4
```

This is equivalent to using the command `max(size(X))`.

1.4.4 The disp() Function

The `disp()` function writes a string to the MATLAB Command Window output, as shown in the following listing:

```
>> some_str = 'hello';
>> disp(some_str)
hello
```

1.4.5 The num2str() Function

The `num2str(x, formatStr)` converts a number (`double`) to a `char` string, with options specified using the string `formatStr`.

If we wish to display the value of a number using `disp()`, we must convert it to a string first. This may be done using (`num2str()`, for example:

```
>> x = 9;
>> disp(['x = ', num2str(x)])
x = 9
```

Since `disp()` receives only one string input argument, we horizontally concatenate the two strings `'x = '` and `num2str(x)` into one using horizontal concatenation.

1.4.6 Obtaining Time Intervals using tic and toc

Have you ever wished that you can use a stopwatch in MATLAB? This can allow you to determine how long some code will take, like starting a stopwatch when a runner leaves the starting line and stopping it when she crosses the 400-m mark. This could also allow you to execute a `while` loop as long as elapsed time is within some limit.

Time intervals may be measured using the `tic` and `toc` commands. When you wish to start the time, use `tic`. A subsequent `toc` stops the timer and returns the time interval measurement; or you can measure a time interval and make a count-down timer. For example:

```
>> tic; % like starting a stopwatch
pause(0.25); % insert a pause to simulate some commands run
time_elapsed = toc % ends the timer and reports the time elapsed

time_elapsed =

    0.2555

>>
```

Using a variable, such as `x = tic`, establishes `x` as a time marker. When supplied as an input to the `toc` command, `toc` returns time elapsed since `x` was defined. Thus, multiple reference points may be used in a calculation. For example, code like this may be used:

```
start_time = tic; % interval start time reference
pause(0.25); % insert a pause to simulate some commands run
first_interval = toc(start_time); % measure first time interval
mid_time = tic; % adds a second reference, mid_time
```

```
pause(0.75);  % insert a pause to simulate some commands run
end_interval = toc(mid_time); % measure time since mid_time
total_time = toc(start_time); % measure time since first_time
disp(sprintf('First interval: %5.3f s', first_interval) )
disp(sprintf('Second interval: %5.3f s', end_interval))
disp(sprintf('Total duration: %5.3f s', total_time))
```

The output would be something like this:

```
First interval: 0.257 s
Second interval: 0.755 s
Total duration: 1.016 s
```

To see more demonstrations of `tic` and `toc`, see this video.

1.5 User-defined Functions

A MATLAB function is typically defined in a text file with the extension `*.m`. Here are some salient features of a function definition file.

1. The name of the `*.m` file must match the function name exactly (MATLAB is case-sensitive).

2. The file begins with the keyword `function`, followed by the name of the function. This first line of the function is called the *header*, and it specifies how the user interacts with the function. The syntax of the function header (with end) is given below:

   ```
   function [out1, out2] = functionName( in1, in2, in3 )
     statements
   end
   ```

   Here, the function `functionName` is a three-input, two-output function.

3. The file ends with the keyword `end`.

4. The *body* consists of the statements that specify the function implementation. The body is written between the header and the closing `end` statement. There may be only very few statements in the body, or several hundreds or even thousands of lines of code in a function body.

5. The function header defines the inputs required by the function, as well as the outputs provided by the function.

   (a) Functions may be designed with no input or output, or few inputs and outputs, or even variable inputs and outputs.

   (b) Functional inputs and outputs are called **arguments** or **parameters**. The parameters defined in the header are to be used within the function body.

6. Commented lines of code immediately following the header provide function help documentation. When you type `help functionName` in the command line, the function help you defined appears in the command line.

Example 1.1. Example Write and test a function that calculates $f(x, y, z)$, where

$$f(x, y, z) = x + y^2 + z^3.$$

Solution. Here, there are three input parameters: x, y, and z; and one output parameter f. Thus, we can accomplish this by writing the following function and saving it in the text file `f_function.m` of Listing 2:

```
function f = f_function(x, y, z)
%f_function calculates f(x, y, z), where
%    f(x,y,z) = x + y^2 + z^3
%
% By E.P. Blair
% Baylor University
%
f = x + y^2 + z^3;
end
```

Listing 2: The code of the function-definition file `f_function.m`.

To test `f_function.m`, we can write a *testbed* script, or we can simply try it in the command line. A testbed script is one which invokes a function of interest in order to test whether it performs as designed. Here, however, we simply test `f_function()` in the MATLAB command line:

```
1  >> f = f_function(1, 1, 1)
2
3  f =
4
5       3
6
7  >> f = f_function(1, 2, 3)
8
9  f =
10
11     32
```

Listing 3: MATLAB Command Window input invoking `f_function()`, along with the resulting output.

Line 1 of Listing 3 invokes `f_function()` with $x = y = z = 1$, which returned the correct result: $f(1, 1, 1) = 1 + 1^2 + 1^3 = 3$. This matches the output of lines 3-5. The next invocation of `f_function()` correctly evaluated $f(1, 2, 3) = 1 + 2^2 + 3^3 = 1 + 4 + 27 = 32$, shown in the result of lines 9-11. Thus, `f_function()` appears to work correctly.

Functions may also take data of type `char`, `cell`, `struct`, or even objects of classes as input. Outputs may be of the same types.

1.5.1 The `varargin` Keyword

The key word `varargin` may be used in a function header to allow a variable number of input arguments. For example, in the function definition below, inputs `in1`, `in2`, and `in3` are mandatory, but `varargin` allows for zero or more additional input arguments.

```
function [out1, out2] = functionName( in1, in2, in3, varargin)
  statements
end
```

Thus, through the flexibility afforded by `varargin`, the following invocations of `functionName` all are valid:

```
[a, b] = functionName( x, y, z );

[c, d] = functionName( x, y, z, e );

[g, h] = functionName( x, y, z, e, j, k );
```

Of course, we must design the body of `functionName()` to correctly support the various invocations.

1.5.2 The `nargin` Function

MATLAB has a special function named `nargin`. When invoked inside another function, `nargin` returns the number of input arguments specified in an invocation of the function of interest. This is particularly useful when that function is designed to support variable input arguments using `varargin`. Here, we may use `nargin` with some logical control structure, such as `if-else` or `switch-case`.

You may already have used `varargin` this without knowing it if you have used the `plot()` command:

```
x = linspace(0, 1, 101);
y = sin(2*pi*x);
figure;
plot(x, y); % generates a default plot
figure % new figure
plot(x, abs(y), 'LineWidth', 2', 'LineStyle', '--'); % new plot,
    this time with options
```

Notice that we use the same `plot()` function, but with a different number of arguments. The key to doing this is `varargin`, and now that key is in your hand!

For example, here is a function that allows a user to calculate values of a quadratic or a cubic expression:

```
function y = calc_poly(x, a, b, c, varargin)
%CALC_POLY calculates the value of a quadratic, cubic or quartic
%         expressions.
%
% SYNTAX
% ======
%   y = calc_poly(x, a, b, c ) evaluates the value of the quadratic
%       expression a*x^2 + b*x + c
%
%   y = calc_poly(x, a, b, c, d ) evaluates the value of the cubic
%       expression a*x^3 + b*x^2 + c*x + d
%
```

```
%   y = calc_poly(x, a, b, c, d, f ) evaluates the value of the
%         quartic expression a*x^4 + b*x^3 + c*x^2 + d*x + f
%
% By. E.P. Blair
% Baylor University
% 2021.04.17

switch nargin % number of input arguments
    case 4 % quadratic
        y = a*x.^2 + b*x + c;

    case 5  % cubic
        d = varargin{1};
        y = a*x.^3 + b*x.^2 + c*x + d;

    case 6  % quartic
        d = varargin{1};
        f = varargin{2};
        y = a*x.^4 + b*x.^3 + c*x.^2 + d*x + f;

    otherwise
        error(['calc_poly: invalid number (', num2str(nargin), ...
            ') of input arguments.'])

end
```

1.6 Classes

1.6.1 Beyond Structures

Sometimes, simply storing information in structures is not enough. In these cases, it is desirable to perform manipulations on the various groups of information, or model the effects of particular events on the items represented by structures. In these cases, it is powerfully helpful to define **classes**. Classes extend the capability of `structs` by defining a standard set of fields, called **properties**, and by defining an associated set of class **methods**. A particular instance or extended `struct` of a class is called an **object**. We can think of an object as a variable of a custom-data type (the class). Methods—sometimes called *behaviors* in other programming languages —are a set of functions that are used to extract information from or manipulate objects.

1.6.2 Motivation for Classes

One example where classes might be useful is in an online gaming system. Here, we might want a class called `Avatar` (we will use the convention in which we capitalize the name of a user-defined class to distinguish it from MATLAB's own pre-defined classes). For each individual player, an object of class `Avatar` may store the user's real name, handle, level, experience points, maximum vitality, and health. Then, `Avatar` class methods can define operations on objects such as `gainXP()` to add to a player's experience points, `levelUp()` to implement an irreversible milestone in the development of the player's avatar, and `attack()` to model one player's attack on another player, which may detract from the health of the target of the attack. One can imagine myriad other methods that might be desirable in a complex gaming system.

Another example in which classes may be useful is in the design of a software system that tracks an individual's investments. Investments may be in stocks and mutual funds. We might wish to make an `Asset` class that stores in its member data an asset name, a symbol, and based on a list of transactions, can calculate the worth of that particular holding. A transaction can be represented by objects of a `Transaction` class, which stores information about the transaction date, the type of transaction(open, buy, sell, split, dividend, short, close out, etc.), the number of shares transacted, the price per share, and any transaction fees. A particular `Asset` object may include a list of transactions. With the list of transactions and up-to-date information about share price, an `Asset` class method, say, `calculateValue()`, may be used to calculate the worth of the holding. Then, a `Portfolio` class may be designed to contain multiple `Asset` objects. The `Portfolio` class, then, is called a **container** class for `Asset` objects.

1.6.3 The Advantages of Classes

One might think that classes add complexity to computer programs. Indeed, classes and OOP allow vast complexity to be handled in a clean and logical manner. Some of the benefits of using classes:

1. Encapsulation. Many lines of code—think hundreds or thousands—required to implement an operation on an object may be cleanly invoked with a simple call to a method function. Also, if we must modify the method, it can be done once in the class definition. This "under-the-hood" modification may be transparent to a user, who can invoke the modified function using the same syntax as before, but with the benefits of an improved method.

2. Understandability. With well-chosen, smartly-defined methods, OOP adds great understandability. Complex tasks can be executed by invoking aptly-named methods.

3. Hierarchical methods. The user can invoke a method a container-class object, and the container-class method can automatically invoke a method of the objects contained therein. For example, a `Portfolio`-class method `calculateValue()` can invoke the `calculateValue()` for each asset therein. It can gather the returned values and sum them, to report the value of the investment portfolio it represents. This complex method is transparent to the user, who simply queries the `Portfolio`-class object for its value.

2 Built-in Classes

We will gain some experience with classes with MATLAB's built-in classes. A class defines a composite data type. When we define `n = 71.9`, we say that `n` is a variable of type `double`. Similarly, a variable belonging to some class is called an **object**. You'll also hear it said that we *instantiate* an object of a certain class, or that some object is an instance of some class. If we were to write an analogy, we might write `type:variable::class:object`. We can say that a class is a generalization of the concept of a variable.

Like a `struct`, an object may have multiple elementary data types embedded in it. In a `struct`, the constituent data elements are called "fields," but in an object the constituent data elements are called "properties."

2.1 The `datetime` Class

The `datetime` class is provided to allow users to create and manipulate dates.

2.1.1 Creating a New `datetime` Object

We can instantiate a `datetime` object, `x`, as follows:

```
>> x = datetime('now')

x =

  datetime

   17-Apr-2021 19:44:34

>> class(x)

ans =

    'datetime'
```

Listing 4: This input to the MATLAB Command Window input uses the `datetime()` constructor method to instantiate a `datetime` object, `x`.

MATLAB automatically reports that this is a `datetime` object, and the `class` function is in agreement with this.

2.1.2 Properties

Building on the previous code block, we can then use the `properties()` function to discover what properties the object `x` has. Then, we can reference those properties using the dot syntax:

```
>> properties(x)

Properties for class datetime:

    Format
    TimeZone
```

```
    Year
    Month
    Day
    Hour
    Minute
    Second

>> x.Year

ans =

        2021

>> disp(['Month: ', num2str(x.Month)])
Month: 4
```

Listing 5: The `properties()` function lists the accessible properties of the `x` object.

2.1.3 Method Functions

Any function associated with a class is called a **method function**, or simply a “method.” Methods define how objects can be used and manipulated, and how data contained in functions can be accessed.

The `datetime` function is actually a special method called a **constructor method**. This defines how a new object is instantiated for the class. In the previous code block, we use the string `'now'` as an input to the constructor `datetime`. This caused `datetime` to obtain the current computer clock time and form a `datetime` object. The `datetime` constructor may also be used to create a `datetime` object for a specific day:

```
>> bday = datetime(2016, 10, 5)

bday =

  datetime

   05-Oct-2016

>> age = between(bday, datetime('now'))

age =

  calendarDuration

   4y 6mo 13d 0h 24m 30.366s
```

Listing 6: A `datetime` object is created to represent October 5, 2016 as a birthday.

You can use `help datetime` to discover other ways to use the `datetime` constructor.

Other methods can be defined to access, display or manipulate the data in an object. For example, `between` is a `datetime` function designed to find the duration between two `datetime` objects, as in lines 9-15 of the above listing.

2.2 MATLAB Graphics Classes

If you've used MATLAB graphics, you've already used classes and objects, whether you realize it or not. MATLAB graphics make heavy use of object-oriented programming.

2.2.1 The `Figure` Class

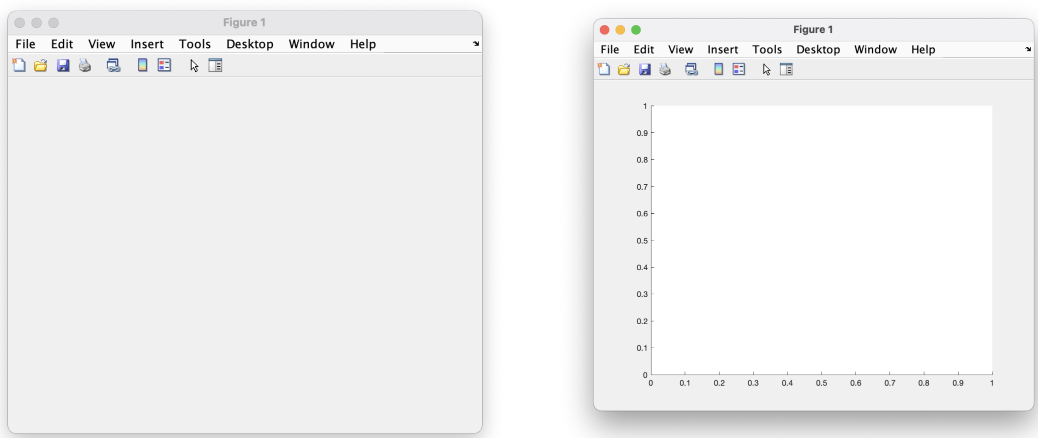
Consider the following Command Window session, which creates a new figure:

```
>> myFig = figure

myFig =

  Figure (1) with properties:

      Number: 1
        Name: ''
       Color: [0.9400 0.9400 0.9400]
    Position: [617 599 560 420]
       Units: 'pixels'

  Show all properties

>> ishandle(myFig)

ans =

  logical

   1
>> m = 7; ishandle(m)

ans =

  logical

   0
>> myFig.Position

ans =

   617   597   560   420

>>
```

Listing 7: A new figure is created in the MATLAB Command Window.

This creates the blank figure depicted in Fig. 1(a). The Command Window output reports that the variable `myFig` actually is an object of class `Figure`. It also lists some of the most important properties of `myFig`, and you could see more properties by clicking on `all prooperties` in the Command Window.

The variable `myFig` is a special kind of variable called a **handle**. A handle is a variable that allows us to reference a MATLAB graphics object (objects of class `Figure`, `Axes`, `Line`, etc.), and line 15 demonstrates the use of `ishandle()` to verify `myFig` as a handle. Line 22 also demonstrates what happens when the argument to `ishandle` is a non-handle variable. Handles are useful if we wish to manipulate graphics objects using MATLAB code. Line 29 demonstrates how we might obtain the properties of `myFig` for future use. Here, we have obtained the (x, y)-coordinate of the lower-left corner of the figure, as well as its width and height, with all units in pixels. We also could adjust the figure position by reassigning these values using dot syntax, with a command like `myFig.Position = [x y w h]` for some previously-defined `x`, `y`, `w`, and `h`.



(a) (b)

Figure 1: (a) A new, blank figure. (b) A new, blank axes is added to the figure.

A subsequent use of a command like `newerFig = figure` command will create a new figure with number 2, and this figure may be referenced using the handle `newerFig`. When the `figure()` command is invoked with a handle to an existing figure, such as `figure(someHandle)`, the figure referenced by `someHandle` is brought to the top, shown in front of all other MATLAB figures, and it is designated as the **current figure**. Subsequent plotting commands will occur in the current figure. The current figure also may be referenced by the command `gcf`, which is short for "get current figure."

2.2.2 The `Axes` Class

If we continue the Command Window session from the previous listing, we can add a blank axes to the pre-existing figure as follows:

```
>> myAx = axes

myAx =

```

```
  Axes with properties:

              XLim: [0 1]
              YLim: [0 1]
            XScale: 'linear'
            YScale: 'linear'
     GridLineStyle: '-'
          Position: [0.1300 0.1100 0.7750 0.8150]
             Units: 'normalized'

  Show all properties

>>
```

Listing 8: A new, blank axes is added to the figure.

Note that if `myAx` is embedded in `myFig`, an entry for this axes is made in the `myFig.Children` property:

```
>> myFig.Children  % this points to an Axes object

ans =

  Axes with properties:

              XLim: [0 1]
              YLim: [0 1]
            XScale: 'linear'
            YScale: 'linear'
     GridLineStyle: '-'
          Position: [0.1300 0.1100 0.7750 0.8150]
             Units: 'normalized'

  Show all properties

>>
```

Listing 9: When `myAx` is embedded in `myFig`, the `myFig.Children` property also may be used to reference the new axes object.

The new axes also may be referenced using `myFig.Children`.

When multiple subplots are defined within the current figure, the `Children` property of the current axes is an array:

```
>> plot_a = subplot(2,1,1) % create an upper subplot

plot_a =

  Axes with properties:

              XLim: [0 1]
```

```
              YLim: [0 1]
            XScale: 'linear'
            YScale: 'linear'
     GridLineStyle: '-'
          Position: [0.1300 0.5838 0.7750 0.3412]
             Units: 'normalized'

  Show all properties

>> plot_b = subplot(2,1,2) % create a lower subplot

plot_b =

  Axes with properties:

              XLim: [0 1]
              YLim: [0 1]
            XScale: 'linear'
            YScale: 'linear'
     GridLineStyle: '-'
          Position: [0.1300 0.1100 0.7750 0.3412]
             Units: 'normalized'

  Show all properties

>> myFig.Children % now the figure has a two child Axes objects

ans =

  2x1 Axes array:

  Axes
  Axes

>> myFig.Children(1) % points to the first Axes in the array

ans =

  Axes with properties:

              XLim: [0 1]
              YLim: [0 1]
            XScale: 'linear'
            YScale: 'linear'
     GridLineStyle: '-'
          Position: [0.1300 0.1100 0.7750 0.3412]
             Units: 'normalized'

```

```
  Show all properties

>> myFig.Children(2) % points to the second Axes in the array

ans =

  Axes with properties:

             XLim: [0 1]
             YLim: [0 1]
           XScale: 'linear'
           YScale: 'linear'
    GridLineStyle: '-'
         Position: [0.1300 0.5838 0.7750 0.3412]
            Units: 'normalized'

  Show all properties

>>
```

*Listing 10: Multiple **`Axes`** objects are added to **`myFig`**. **`MyFig.Children`** is an array, and each subplot now may be referenced as an element of **`MyFig.Children`** or by any handle associated with that subplot.*

Here, the top subplot is referenced by both `MyFig.Children(1)` and `plot_a`; and, `MyFig.Children(2)` and `plot_b` both are handles for the bottom subplot.

2.2.3 The `Line` Class

When we plot data, we make `Line` objects. The following command plots two lines in one axes:

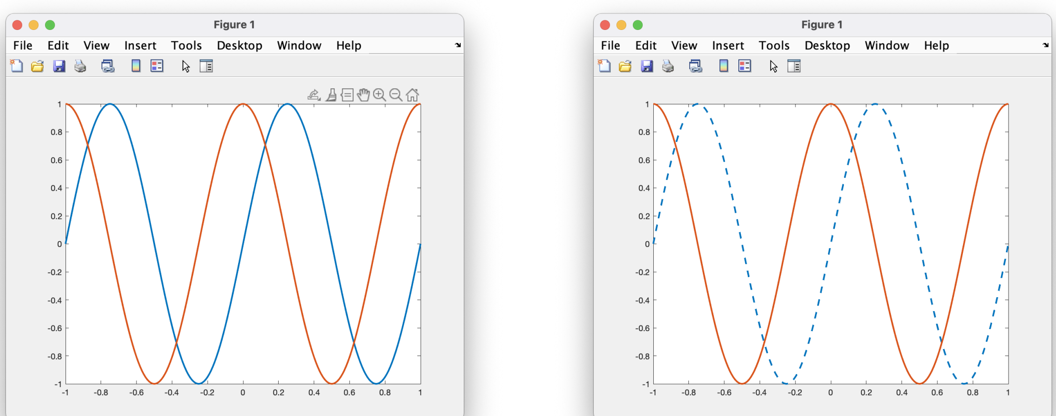
```
>> x = linspace(-1, 1, 201);
% Plot two lines, with data_plot as a reference to the lines
data_plot = plot(x, sin(2*pi*x), x, cos(2*pi*x), 'LineWidth', 2)

data_plot =

  2x1 Line array:

  Line
  Line

>> thisAx = gca; thisFig = gcf; thisAx

thisAx =

  Axes with properties:

             XLim: [-1 1]
             YLim: [-1 1]
           XScale: 'linear'
```

```
21               YScale: 'linear'
22        GridLineStyle: '-'
23             Position: [0.1300 0.1100 0.7750 0.8150]
24                Units: 'normalized'
25
26   Show all properties
27
28 >> thisAx.Children % the children of thisAx are Line objects
29
30 ans =
31
32   2x1 Line array:
33
34   Line
35   Line
36
37 >>
```

*Listing 11: Two data sets are plotted on one **Axes** object.*

This code produced the `Figure`, `Axes`, and plot of Figure 2. The result is two line objects, which may be referenced in two ways: (1) using the elements of the `data_plot` array [lines 5-10]; and (2) as children of the `Axes` object [lines 29-36]. Additionally, we obtain handles (line 28) to the current axes (`thisAx`) and current figure (`thisFig`) using `gca` (short for *get current axes*) and `gcf` (short for *get current figure*).

Then, we may also reference the two lines using `thisAx.Children`. Recall that the `Axes` object is a child object to the `Figure` object, contained in the `Figure` property `Children`. The `Line` objects, however, are listed as children in the `Children` property of the `Axes` object `thisAx`.



(a) Initial plot. (b) After setting `data_plot(1).LineStyle = '--'`.

*Figure 2: Two data sets are plotted in one **Axes** object.*

Let us inspect the properties of the `Line` object referenced by `data_plot(1)`:

```
1  >> data_plot(1) % get properties of
2  
3  ans =
4  
5    Line with properties:
6  
7                Color: [0 0.4470 0.7410]
8            LineStyle: '-'
9            LineWidth: 2
10              Marker: 'none'
11          MarkerSize: 6
12     MarkerFaceColor: 'none'
13               XData: [1x201 double]
14               YData: [1x201 double]
15               ZData: [1x0 double]
16  
17    Show all properties
18  
19 >> data_plot(1).LineStyle = '--';
```

Listing 12: The properties of `data_plot(1)`.

Line 19 shows how we can use the dot syntax to change the `LineStyle` property for `data_plot(1)`, resulting in Figure 2(b).

The `Line` object has special properties, `XData`, `YData`, and `ZData`. These contain the actual x, y, and z coordinates of data points plotted. Since there is no `ZData`, the plot is a 2D plot. If you wish to change or animate your plot, it could be faster and visually smoother to simply update these properties, rather than replot the data with a new call to the `plot()` command. This will save time, since you do not have to clear the axes nor reformat it if you only adjust these special properties of the `Line` object.

An alternate and older way to get MATLAB graphics properties is to use the `get()` function. For example, `line_col = get(data_plot(1))` stores the red-green-blue (RBG) triple for `data_plot(1)` in `line_col`. This may also be done for any object of a MATLAB graphics class (`Figure`, `Axes`, `Line`, `Patch`, etc.).

Similarly, we may use the `set()` function to set properties of a MATLAB graphics object. While the dot syntax allows only one property to be modified at a time, the `set()` function allows multiple properties to be adjusted in one command. For example, we could use two commands to adjust the line style and line marker for `someLine`:

```
>> someLine.LineStyle = '--';
>> someLine.Marker = 'o';
```

This may be done in one call of the `set()` function:

```
>> set(someLine, 'LineStyle', '--', 'Marker', 'o');
```

2.2.4 The `Patch` Class

A MATLAB `Patch` object is a drawing of a closed polygon with straight edges. The polygon is specified by the the x-, y-, and (optional) z coordinates of each vertex. The polygon is closed by connecting the last vertex to the first. Thus, it is redundant to have the first vertex and the last vertex as the same coordinate.

The basic syntax for the `patch()` function is `patch(X,Y,C)`. This creates a polygon of N points, where `X` is a $1 \times N$ vector of x values, Y is a $1 \times N$ vector of y values, and `C` is a color specifier. The color specifier can be a string to specify a basic color, such as `'r'`, `'g'`,`'b'`,`'c'`,`'m'`,`'y'`,`'w'`, or `'k'`; or, `C` may be a 3-element RGB (red-blue-green) triple specifying an arbitrary color. For example: `C = [1, 0, 0]` specifies elementary red; `C = [0, 1, 0]` specifies green; `C = [0, 0, 1]` specifies blue; `C = [0, 0, 0]` specifies black; and `C = [1, 1, 1]` specifies white.

As an example, I provide the listing of a `basicPatch.m` script. The patch is defined in lines 3-7, and following lines of code provide formatting.

```
1  % basicPatch.m
2
3  x = [-1 -1 1 1]; % specify x-values for vertices
4  y = [-1 1 1 -1]; % specify y-values for vertices
5  C = [0.75 0 0.75]; % specify a purple-ish color
6
7  newSquare = patch(x, y, C)
8  set(gca, 'FontName', 'Times', 'FontSize', 20); % format the current
       axes
9  grid on;
10
11 xlim([-3 3]); % adjust the x-limits of the axis
12 ylim([-3 3]); % adjust the y-limits of the axis
13
14 xlabel('$x$ (m)', 'Interpreter', 'latex') % add an x-label
15 ylabel('$y$ (m)', 'Interpreter', 'latex') % add a y-label
```

Listing 13: Listing of the script `basicPatch.m`.

The output of the `basicPatch.m` script is shown in Fig. 3.

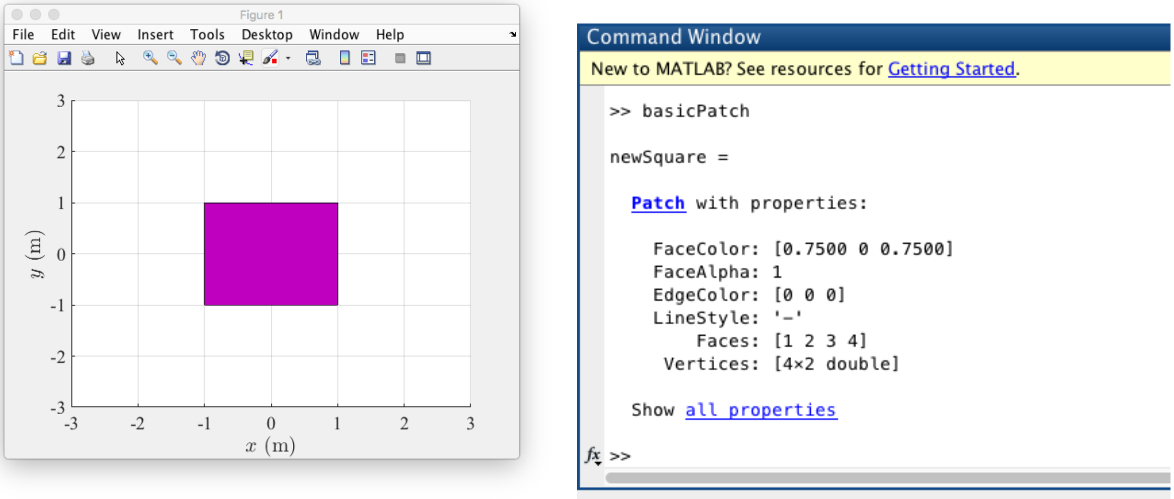
The `Patch` object is centered at the origin in Fig. 4(b). An example of a simple modification to this is to translate the patch by the vector $(\Delta x, \Delta y)$. To do this, we define Δx and Δy, and add these to the original `XData` and `YData` properties of the `newSquare` object. This can be done in the Command Window using the `set()` function:

```
1  >> Dx = -2; Dy = 1.5;
2  >> set(newSquare, 'XData', newSquare.XData + Dx, 'YData', newSquare.
       YData + DY)
```

Listing 14: A Command Window modification to the `patch` defined in Listing 13. Line 1 is used to specify the x- and y-components of the displacement. The displacement is then applied to the original `XData` and `YData` properties, and the result of the addition is the value component of a property-value pair in the `set()` function.

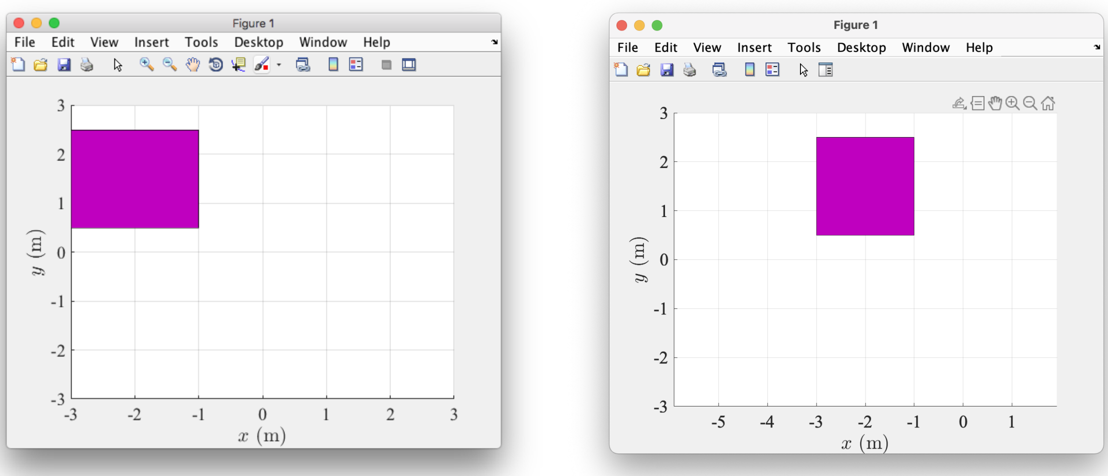
The syntax here is `set(obj, Prop1, Val1, Prop2, Val2, ...)`, where `obj` is the object we wish to modify, and we use property-value pairs to assign new object properties. The displaced `patch` object is displayed in Fig. 4(a). Similar modifications may be made to a `line`-class object.

You may have noticed that the aspect ratio is not equal for the plot of Figure 4(a). This may be improved using the `axis()` function:



(a) (b)

Figure 3: Graphical and Command Window output of the `basicPatch.m` *script of Listing 13.*



(a) (b)

Figure 4: (a) The `patch` *object created using* `basicPatch.m` *is modified by using the* `set()` *function using the code of Listing 15. The resulting* `patch` *no longer sits at the origin. (b) The axes aspect ratio is made to be equal.*

```
1 >> axis(gca, 'equal')
```

Listing 15: A Command Window modification to the `patch` *defined in Listing 13. Line 1 is used to specify the x- and y-components of the displacement. The displacement is then applied to the original* `XData` *and* `YData` *properties, and the result of the addition is the value component of a property-value pair in the* `set()` *function.*

This may adjust the x-limits and y-limits of the plot or only the positioning of the `Axis` object to get an equal aspect ratio.

For more information about the `patch` class, type `doc patch` in the MATLAB Command Window, or do an Internet search.

2.2.5 Animations using MATLAB Handle Graphics

When making animations in MATLAB, we may (1) replot each frame, or we may simply (2) adjust `XData`, `YData`, and `ZData` for objects in the plot for each frame. Method 2 is to be preferred, especially for interactive animations, because it is faster and avoids reconstructing and reformatting the plot. This results in a smoother experience for the user. It also may be helpful to set the axes limits using `xlim` and `ylim` so that relative positions of graphical objects is readily apparent between frames and the graphics for each frame all are plotted against the same background.

For an animation example and tutorial, see this demonstration on animating "rain drops" in MATLAB.

3 Defining Classes in MATLAB

A class is a composite data type. MATLAB has several built-in classes, such as `figure`, and numerous graphics classes. MATLAB also allows you, the user, to define your own classes. To do so, you must write a class definition.

A class definition is a set of instructions defining for MATLAB your custom data type. A variables of a custom type is called an **object**.

3.1 The Class-definition File

In MATLAB, we define a class using a class-definition file. The file name must be identical to the name of the class itself, and it must have the '`.m`' extension. For example, to define an `Asset` class, we create a file named '`Asset.m`' (without the single quotes, of course).

A class definition file may be obtained by the '`New Document`' button on the `Home` tab of the MATLAB IDE and selecting the '`Class`' option. This opens a new editor window with a textual template for your new class. An example of a new class template is shown in Fig. 5. Alternately, one can simply select a new script and write the class from scratch.

Let us examine some of the features of the template class file generated for us by MATLAB 2017b (see Fig. 5):

1. The class definition files begins the keyword `classdef`, followed immediately by the class name (line 1). Also, lines of comments immediately follow line 1, providing help documentation for the class. Together, the documentation comments and line 1 provide a class header.

    - In this case, the class template has the text `untitled5` as a placeholder for the class name.

2. Following the class header, there are two sections to the class definition:

    - The **properties section** begins with the key word `properties` (line 5) and ends with the key word `end` (line 7).

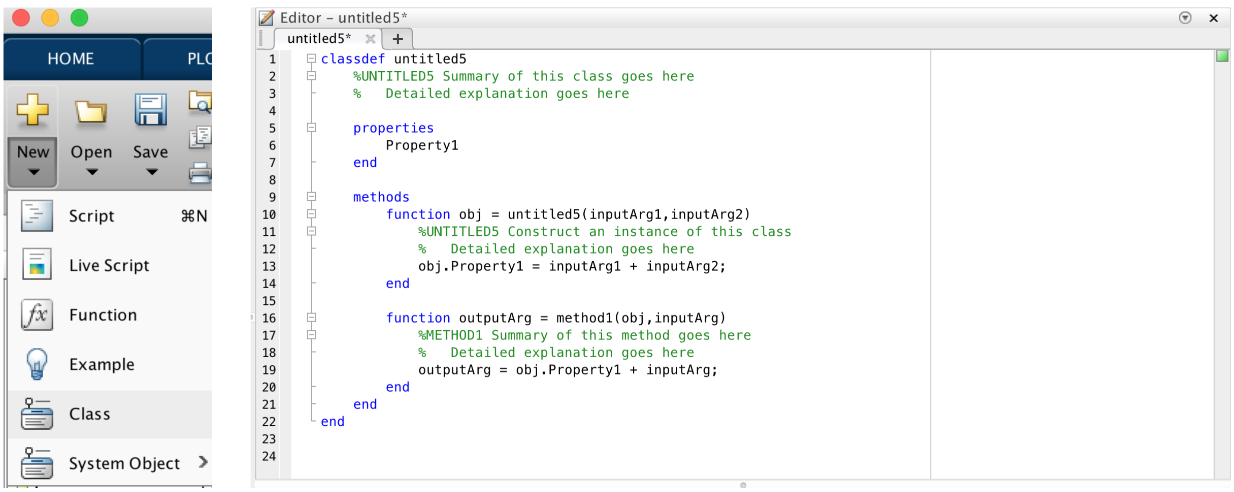
      The body of the `properties` section is used to define class properties (also known as "fields", or "member data"). Here, there is one dummy property defined: `Property1`.

    - The **methods section** begins with the key word `methods` (line 9) and ends with the key word `end` (line 21).

      The body of the `methods` section is used to define functions which operate on objects. Such functions are more precisely called **methods**.

        – The first method is an important function known as a **constructor** method. The constructor shares exactly the same name as the class, and is used to create or define an object of the class. A constructor typically receives input arguments which are used to specify some object properties. Typically, a constructor has a single output, `obj`, which is the desired result of the constructor. Of all the class methods, the constructor is unique in the sense that it does not operate on nor is it associated with an existing object. All other method functions operate on at least one object, and thus require an object as the first input argument, typically `obj`.
          In particular, the placeholder constructor method `untitled5()` sums input arguments `inputArg1` and `inputArg2` and stores the result in the lone object property `Property1`.

– The second method, `method1()`, is of the typical, non-constructor form. The first input argument is an object, `obj`. Within a non-constructor method, the object `obj` is only a copy of the object passed to the function in the first argument. In particular, `method1()` sums `inputArg` and the object property `obj.Property1` and returns this result as `outputArg`. This function does not modify the original object. In fact, within `method1()`, the code actually works with a copy of the object specified by `obj`.



(a) (b)

Figure 5: (a) The "New Document" button provides a "Class" option. (b) A new class template obtained in MATLAB 2017b.

Example 3.1. Example Create an object `myObj`.
Solution.
To do this, we can invoke the constructor `untitled5()` using the syntax:

```
1 >> x1 = 1; x2 = 2; myObj = untitled5(x1, x2)
2
3 myObj =
4
5   untitled5 with properties:
6
7     Property1: 3
```

Listing 16: MATLAB Command Window input utilizing and testing the `untitled5` *class definition of Fig. 5.*

Here, the command-line input of line 1 assigns the values 1 and 2 to `x1` and `x2`. Those values, then, are passed to the `untitled5()` constructor method to instantiate the object `myObj` of class `untitled5`. The output of this command is seen in lines 3-7, in which we see that `myObj` property `Property1` stores the value 3, which is the sum of the values stored in `x1` and `x2`.

Example 3.2. Invoke the method `method1()` on the object `myObj` created in Example 3.1.
Solution.
To do this, invoke `method1()` using the syntax:

```
1 >> x = myObj.method1(5)
2
3 x =
4
5      8
```

To invoke `method1()` on `myObj`, we use the dot (".") syntax `myObj.method1(...)`, much like a reference to a property of `myObj`. This syntax is unique to class methods. While `myObj` does not appear within the input argument list to `method()` in line 1 above, it is in fact the first argument to `method()` because of the dot syntax. An alternative and equivalent—but older and depreciated—syntax for invoking `method1()` in line 1 above is `x = method1(myObj, 5)`. This syntax explicitly specifies `myObj` as the first argument. The older syntax is used below, with the same result as above.

```
>> x = method1(myObj, 5)

x =

     8
```

3.1.1 Modifying Objects

By default, when we pass an object to a method using either `obj.method1()` syntax or the older syntax, **only a copy of the object `obj` is passed to `method1()`**. Thus, any modifications `method1()` makes to its object are made to the *copy*, not the original object.

If we wish to make a method `method2()` that modifies an object `myObj`, we need to define `method2()` in the following manner:

```
function obj = method2(obj, InputArg1, InputArg2, ... )
      < statements that modify obj >
end
```

Now, `method2()` returns the modified copy of the input object. The syntax to modify `myObj` is as follows:

```
myObj = myObj.method2( a, b, ... );
```

This invocation of `method2()` allows `method2()` to modify a copy of `myObj` inside the Workspace of `method2()`. Then, `method2()` returns a modified copy of `myObj`, which is used to overwrite the original, unmodified object `myObj`.

4 Extended Example: Developing Software to Track Investments

4.1 Developing an `Asset` Class

Now we begin an extended, multi-part example, which begins with an `Asset` class. We will initially define an `Asset` class with the following properties: `name`, `symbol`, `quantity`, `units` and `transactionList`. Here `name` can store a `char` string to identify a company or mutual fund, and `symbol` can store a stock ticker symbol or equivalent symbol as a `char` string. The `units` property can store a `char` string specify the units of this asset: `'shares'`, `'USD'`, `'RMB'`, etc. Finally, the `transactionList` will store the list of transactions pertinent to the holdings in this asset. A transaction will be modeled as an object of the `Transaction` class, yet to be defined.

To do this, we prepare the class definition file `Asset.m`:

```
1  classdef Asset
2      %Asset defines an Asset class to store information about a
          particular
3      %investment asset (stock, mutual fund, currency, etc.).
4      %
5      % By E.P. Blair
6      % Baylor University
7
8      properties
9          name
10         symbol
11         units
12         transactionList
13     end
14
15     methods
16         function obj = Asset(AName, ASymbol, AUnits)
17             %Asset Constructs an instance (obj) of the Asset class
18             %   Here, the required syntax is
19             %   >> myObj = Asset( nameStr, symbolStr, unitStr )
20             %   where nameStr, symbolStr, and unitsStr are char
                  strings
21             %   specifying the asset name, symbol, and units ('
                  shares',
22             %   'USD', 'RMB', etc.).
23             obj.name = AName;
24             obj.symbol = ASymbol;
25             obj.units = AUnits;
26         end
27
28     end
29 end
```

Here, in the `properties` section, lines 9-12 establish the desired class properties. The constructor of lines 16-26 is designed to accept three inputs: `AName`, `ASymbol`, and `AUnits`. The value of these parameters are then stored in the appropriate fields of the `Asset` object.

To test the `Asset` class definition, we can create a testbed script named `testbedAsset.m`:

```
% testbedAsset.m

newAsset = Asset('X-ray Yankee Zulu, Inc.', 'XYZ', 'shares')
```

This script has one line, which simply invokes the `Asset` constructor to instantiate a new object, `newAsset`. The result of this testbed is:

```
1 >> testbedAsset
2
3 newAsset =
4
5   Asset with properties:
6
7                 name: 'X-ray Yankee Zulu, Inc.'
8               symbol: 'XYZ'
9                units: 'shares'
10     transactionList: []
```

Lines 2-10 list the output. Here, the required method inputs populate the appropriate object properties, and the `transactionList` remains empty.

4.2 Developing a `Transaction` Class

Now we will continue to develop the extended example started in Sect. 4.1 by developing a `Transaction` class to have the following properties: `date`, `transactionType`, `quantity`, `price` and `fees`. Here, `date` can store an object of the pre-defined MATLAB class `datetime`; `transactionType` can store a `char` string to identify whether a transaction is a `'buy'`, `'sell'`, `'short'`, etc.; `quantity` and `price` can store a numerical value for the number of units transacted and the unit price, respectively; and, `fees` can be used to store transaction fees.
Solution.

```
classdef Transaction
    %Transaction defines a Transaction class to represent a transaction of
    %an investment assset, represented by the Asset class.
    %

    properties
        date % a MATLAB datetime object
        transactionType % 'buy', 'sell', 'short', 'dividend', 'split'
        quantity
        fee
        price

    end % END: properties

    methods
```

```
        function obj = Transaction( varargin )
            %Transaction constructs an instance of the Transaction class.
            %
            % SYNTAX:
            %
            % newTrans = Transaction( Ttype, Tquantity, Tprice)
            %     Defines a new transaction with the current date/time in
            %     the date field.
            %
            % newTrans = Transaction( Tdate, Ttype, Tquantity, Tprice)
            %     Defines a new transaction with a specified date/time. The
            %     date/time object may be specified as MATLAB datetime
            %     object or as a 3- or 6-element date vector.
            %
            % newTrans = Transaction( Tdate, Ttype, Tquantity, Tprice, fee)
            %     Defines a new transaction with a specified date/time and
            %     a transaction fee.
            %

            % This switch-case control group defines the Transaction object
            % differently based on the
            switch nargin
                % The 3-input case assumes that the date is now
                case 3 % newTr = Transaction(Ttype, Tquantity, Tprice)
                    obj.date = datetime('now');
                    obj.transactionType = varargin{1};
                    obj.quantity = varargin{2};
                    obj.fee = 0;
                    obj.price = varargin{3};

                % The 4-input case allows the transaction date to be
                % specified
                case 4 % newTr = Transaction(Tdate, Ttype, Tquantity, ...
                    %               Tprice)

                    switch class(varargin{1})
                        case 'datetime'
                            obj.date = varargin{1};
```

```
                        case 'double' % the date specifier is in vector form
                            % convert a date vector to a datetime object
                            obj.date = datetime(varargin{1});
                        otherwise
                            error('Invalid transaction date specification.')
                    end

                    obj.transactionType = varargin{2};
                    obj.quantity = varargin{3};
                    obj.price = varargin{4};

                case 5
                    % newTr = Transaction( Tdate, Ttype, Tquantity, ...
                    %           Tprice, Tfee )

                    switch class(varargin{1})
                        case 'datetime'
                            obj.date = varargin{1};
                        case 'double' % the date specifier is in vector form
                            % convert a date vector to a datetime object
                            obj.date = datetime(varargin{1});
                        otherwise
                            error('Invalid transaction date specification.')
                    end

                    obj.transactionType = varargin{2};
                    obj.quantity = varargin{3};
                    obj.price = varargin{4};
                    obj.fee = varargin{5};

                otherwise
                    error('Invalid number of input arguments.')
            end

        end


    end % END: methods
end
```

The above listing defines the `Transaction` class, with only a constructor. In the `Transaction` constructor, fairly advanced techniques are used, such as a variable set of input arguments. Calls to the constructor are allowed with three, four, or five inputs. In the first (three-input) case, the `date` field is assumed to be the current date and time. The four-input and five-input cases allow the specification of the `date` property as either a date vector or as a `datetime` object (this is a pre-defined MATLAB class). To handle the different input cases, we use the `nargin` function along with a `switch-case` control sequence. A `switch-case` control also is used to enable the flexibility in the specification of the transaction date and time. See Appendix A.3 for more information on `switch-case` controls.

We provide the testbed function `testbedTransaction.m` to test the class:

```
% testbedTransaction.m

% three-input constructor invocation
trans01 = Transaction( 'buy', 75, 71.90 )

% four-input constructor invocation with argument 1 as a date vector
trans02 = Transaction( [2017, 12, 1, 14, 30, 0], 'buy', 100, 7.19 )

% four-input constructor invocation with argument 1 as a datetime
   object
trans03 = Transaction( datetime('now'), 'sell', 25, 52 )

% five-input constructor to specify a transaction fee
trans04 = Transaction( datetime('now'), 'sell', 25, 52, 7 )
```

The output of `testbedTransaction.m` demonstrates that the class definition works as designed:

```
>> testbedTransaction

trans01 =

  Transaction with properties:

               date: 26-Dec-2017 11:13:15
    transactionType: 'buy'
           quantity: 75
                fee: 0
              price: 71.9000


trans02 =

  Transaction with properties:

               date: 01-Dec-2017 14:30:00
    transactionType: 'buy'
           quantity: 100
                fee: []
```

```
            price: 7.1900


trans03 =

  Transaction with properties:

             date: 26-Dec-2017 11:13:15
  transactionType: 'sell'
         quantity: 25
              fee: []
            price: 52


trans04 =

  Transaction with properties:

             date: 26-Dec-2017 11:13:15
  transactionType: 'sell'
         quantity: 25
              fee: 7
            price: 52
```

4.3 Adding Transactions to an `Asset` Object

Now we return to developing the `Asset` class. The goal here is to add transactions to an existing asset, say `someAsset`. We will treat the `transactionList` property as an array of type `Transaction`. If `transactionList` is empty, then the specified transaction is stored in the `transactionList` property. If the `transactionList` is not empty, then the new transaction should be added to the list, and transactions should be listed in chronological order.
Solution.

We add an `addTransaction()` method to the `Asset` class definition file `Asset.m`. For clarity, we will use the notation `ClassName/methodName()` to remove any ambiguity regarding the class with which a method is associated. Also, a method `Asset/listTransactions()` which will compactly list the details of all transactions. This will help us test how well the `addTranstion()` method works. The updated class definition looks like this:

```
classdef Asset
    %Asset defines an Asset class to store information about a
       particular
    %investment asset (stock, mutual fund, currency, etc.).
    %
    % By E.P. Blair
    % Baylor University

    properties
        name = 'X-ray Yankee Zulu'
```

```
        symbol = 'XYZ'
        units = 'shares'
        transactionList
    end

    methods
        function obj = Asset(AName, ASymbol, AUnits)
            %Asset Constructs an instance (obj) of the Asset class
            %   Here, the required syntax is
            %   >> myObj = Asset( nameStr, symbolStr, unitStr )
            %   where nameStr, symbolStr, and unitsStr are char strings
            %   specifying the asset name, symbol, and units ('shares',
            %   'USD', 'RMB', etc.).
            obj.name = AName;
            obj.symbol = ASymbol;
            obj.units = AUnits;
        end

        function obj = addTransaction(obj, newTransaction)
            % addTransaction() adds a transaction newTransaction to the
            % transactionList property of obj

            % add the transaction on the end of the list
            obj.transactionList = [obj.transactionList newTransaction];

            % if the length of the list is greater than 1, the list may
            % require sorting

            if length(obj.transactionList) > 1
                % Create a list of transaction dates by iterating through
                % all transactions and adding dates to dateList
                dateList = []; % empty date list
                for TransIdx = 1:n_trans_old+1
                    % append the date of obj.transactionList(TransIdx) to
                    % dateList (unsorted)
                    dateList = [dateList ...
                        obj.transactionList(TransIdx).date];
                end

                % sort the transaction list

```

```
                % obtain an index of sorted transaction dates
                %   The sort function returns the sorted list along with
                %   the indices of sorted lists within the unsorted list
                %   The indices of the dateList, sortIndex, will be used to
                %   sort the list of transactions.
                [~, sortIndex] = sort(dateList);

                % reorder the unsorted transactions and store in
                % obj.transactionList
                obj.transactionList = obj.transactionList(sortIndex);
            end % END: if length(obj.transactionList) > 1

        end

        function obj = listTransactions(obj)
            numTrans = length(obj.transactionList);

            if numTrans > 0
                for transIndex = 1:numTrans
                    obj.transactionList(transIndex).listDetails;
                end
            else
                disp(['Asset ', obj.name, ' (', obj.symbol, ...
                    ') has no transactions.'])
            end

        end


    end % END: methods
end
```

The new `Asset/addTransaction()` method is the first non-constructor method we've added to the `Asset` class. It works by first appending the new `Transaction` on the end of the `transactionList` property. If the total number of transactions —including the newly-appended transaction —is greater than 1, then the list may require sorting, so we will sort it regardless of whether it requires sorting (it may take even more work to figure out if the list requires sorting). Since the object `obj` is only a copy of the original `obj` upon which `addTransaction()` was invoked, we pass the modified copy `obj` out as an output argument.

The sorting uses the sort command. For a sortable array of elements `x`—such as `doubles` or `datetime` objects, as in the present case—the `[x_sort, sortIndex] = sort(x)` returns a sorted version of `x` in the output `x_sort`, as well as the matching sequence of indices required to sort the original array `x`. This sequence, `sortIndex`, then is used to sort other pieces of data associated with the original array `x`. This is applied in `addTransaction()` when we create `dateList`, an array of `datetime` objects associated with an array of `Transaction` objects (line 45) and subsequently use the `sort()` command on `dateList`. We will not make direct use of a sorted list of dates, so we use ˜ to avoid storing that data in memory within the function `addTransaction`. However, the array `sortIndex` will be used to sort the associated `obj.transactionList` itself.

The `Asset/listTransactions()` method of lines 65-77 will be used to list the details of all `Transaction` objects associated with an `Asset` object. It is designed to use a `for` loop to iterate through all `Transaction` objects, and to print the details of each transaction using a `Transaction` method `listDetails()`, which remains to be defined.

This is an example of hierarchical programming: a user can instruct an `Asset` object to list its transaction details by invoking the `Asset/listTransactions()` method. `Asset/listTransactions()` method, in turn, invokes the `Transaction/listDetails()` method for each associated `Transaction` object. We list below `Transaction` class definition, upgraded with a definition for the `listDetails()` method:

```
classdef Transaction
    %Transaction defines a Transaction class to represent a transaction of
    %an investment assset, represented by the Asset class.
    %

    properties
        date % a MATLAB datetime object
        transactionType % 'buy', 'sell', 'short', 'dividend', 'split'
        quantity
        fee
        price

    end % END: properties

    methods
        function obj = Transaction( varargin )
            %Transaction constructs an instance of the Transaction class.
            %
            % SYNTAX:
            %
            % newTrans = Transaction( Ttype, Tquantity, Tprice)
            %     Defines a new transaction with the current date/time in
            %     the date field.
            %
            % newTrans = Transaction( Tdate, Ttype, Tquantity, Tprice)
            %     Defines a new transaction with a specified date/time. The
            %     date/time object may be specified as MATLAB datetime
```

```
        %     object or as a 3- or 6-element date vector.
        %
        % newTrans = Transaction( Tdate, Ttype, Tquantity, Tprice, fee)
        %     Defines a new transaction with a specified date/time and
        %     a transaction fee.
        %

        % This switch-case control group defines the Transaction object
        % differently based on the
        switch nargin
            % The 3-input case assumes that the date is now
            case 3 % newTr = Transaction(Ttype, Tquantity, Tprice)
                obj.date = datetime('now');
                obj.transactionType = varargin{1};
                obj.quantity = varargin{2};
                obj.fee = 0;
                obj.price = varargin{3};

            % The 4-input case allows the transaction date to be
            % specified
            case 4 % newTr = Transaction(Tdate, Ttype, Tquantity, ...
                %               Tprice)

                switch class(varargin{1})
                    case 'datetime'
                        obj.date = varargin{1};
                    case 'double' % the date specifier is in vector form
                        % convert a date vector to a datetime object
                        obj.date = datetime(varargin{1});
                    otherwise
                        error('Invalid transaction date specification.')
                end

                obj.transactionType = varargin{2};
                obj.quantity = varargin{3};
                obj.price = varargin{4};

            case 5
                % newTr = Transaction( Tdate, Ttype, Tquantity, ...
```

```
                    %            Tprice, Tfee )

                    switch class(varargin{1})
                        case 'datetime'
                            obj.date = varargin{1};
                        case 'double' % the date specifier is in vector form
                            % convert a date vector to a datetime object
                            obj.date = datetime(varargin{1});
                        otherwise
                            error('Invalid transaction date specification.')
                    end

                    obj.transactionType = varargin{2};
                    obj.quantity = varargin{3};
                    obj.price = varargin{4};
                    obj.fee = varargin{5};

                otherwise
                    error('Invalid number of input arguments.')
            end

        end


        function listDetails(obj)
            DateString = [char(obj.date) ...
                blanks(20 - length(char(obj.date)) ) ];
            TypeString = [blanks(10-length(obj.transactionType)), ...
                obj.transactionType];

            QtyString = [blanks(10 - length(num2str(obj.quantity))), ...
                num2str(obj.quantity)];

            RawPriceStr = num2str(obj.price, '%0.3g');
            PriceString = [' at ', blanks(10 - length(RawPriceStr)), ...
                RawPriceStr];

            DetailString = [DateString, TypeString, QtyString, PriceString];

            disp(DetailString)
        end
```

```
108
109     end % END: methods
110 end
```

Additionally, we add some functionality to the `Transaction` class. We add a method `listDetails` that lists the details of a `Transaction` object. The upgraded `Transaction` class is listed in 91-107. Here, we define several strings of fixed width. First, we use the `char` method defined for `datetime` objects to generate a `char` string representing the transaction date (see line 91-92). This string has length `length(char(obj.date))`. We use the `blanks()` function to right-pad this string with white space so that `DateString` is a length of 20 characters always. In line 93, we include the string contained in the `obj.transactionType` property as part of the string `TypeString` but we use the `blanks()` function to left-pad the `obj.transactionType` string with white spaces. This forms `TypeString` as a 10-character string. We use the same technique in line 97 to create a 10-character string detailing the number of units transacted stored in the `obj.quantity` property. Here, the `num2str()` function is used to convert the `double` data representing the number of units transacted to a `char` string. Similarly, lines 100-101 form a fixed-length `char` string `PriceString` detailing the price per unit of the transaction. Finally, in line 104, `DateString`, `TypeString`, `QtyString`, and `PriceString` are concatenated in one string `DetailString`. Then, in line 106, the `disp()` function is used to print `DetailString` to the Command Window output. All of this functionality is called simply within the `Asset listTransactions` method by invoking the `Transaction` class `listDetails` method for each `Transaction` object.

A modified version of `testbedAsset.m` is shown here, in Listing 17:

```
% testbedAsset.m

% create an new Asset object with no transactions
newAsset = Asset('X-ray Yankee Zulu, Inc.', 'XYZ', 'shares')

% add a buy transaction with the current date
newAsset = newAsset.addTransaction( Transaction('buy', 100, 24.03) )

% list transaction data after the first addition
newAsset.listTransactions;

% add a transaction with an earlier date
newAsset = newAsset.addTransaction( Transaction([2017, 12, 1], ...
    'buy', 25, 22.97) )

% list transaction data after the first addition
newAsset.listTransactions;
```

Listing 17: The code listing for `testbedAsset.m`. Here, the testbed adds transactions to `newAsset` and invokes the `listTransactions` method to display information about associated `Transaction` objects.

The output of `testbedAsset.m` is shown below:

```
>> testbedAsset

newAsset =

```

```
5    Asset with properties:
6
7                 name: 'X-ray Yankee Zulu, Inc.'
8               symbol: 'XYZ'
9                units: 'shares'
10     transactionList: []
11
12
13 newAsset =
14
15   Asset with properties:
16
17                name: 'X-ray Yankee Zulu, Inc.'
18              symbol: 'XYZ'
19               units: 'shares'
20     transactionList: [1x1 Transaction]
21
22 Transactions for X-ray Yankee Zulu, Inc. (XYZ):
23 26-Dec-2017 21:36:04       buy       100 at         24
24
25 newAsset =
26
27   Asset with properties:
28
29                name: 'X-ray Yankee Zulu, Inc.'
30              symbol: 'XYZ'
31               units: 'shares'
32     transactionList: [1x2 Transaction]
33
34 Transactions for X-ray Yankee Zulu, Inc. (XYZ):
35 01-Dec-2017                buy        25 at         23
36 26-Dec-2017 21:36:04       buy       100 at         24
```

Listing 18: The output of `testbedAsset.m`

Line 3 of Listing 17 resulted in output lines 3-10. Here, we see that `newAsset` has an empty `transactionList` array property. Line 7 of Listing 17 adds a new transaction, resulting in output lines 13-20 here. This shows that `newAsset` now has one transaction. Line 10 of Listing 17 invokes the `listTransactions()` method for `newAsset`, resulting in output lines 22-23 here. Next, a second, earlier, transaction is added in line 13 of Listing 17. This results in the output of lines 25-32 here. When we again invoke the `listTransactions()` method, we see that not only does `newAsset` have two transactions, but the transactions are listed in chronological order from earliest to latest.

4.4 Calculating the Value of Holdings in an `Asset`

To calculate the value of holdings in an asset, we will add a method `Asset/calculateValue()`. This will iterate through all the `Transaction` objects stored in an `Asset` object's `transactionList`. For each transaction, `Asset/calculateValue()` determine how that transaction will affect the holdings and determine the cost of that transaction based on the type of the transaction and the number of units transacted. In support of this, we first list some upgrades and changes to the `Transaction` class. Changes and upgrades are as follows:

- To support dividend and split transactions, the following class properties are added: `dividend`, `split_ratio`)
- Some properties (`quantity`, `dividend`, `split_ratio`) are assigned a default value. This is done by using an assignment operator and the desired default value along with the property declaration in the `properties` section (syntax: `Property = defaultValue;`).
- The `Transaction` constructor four-input case (inside the `switch nargin` control block) is augmented with a `switch obj.transactionType-case` to handle `div-rnv` (dividend reinvestment) transactions. Here, in the `'div-rnv'` case, the third argument is not the quantity of units transacted, but rather a total dollar amount. The fourth argument remains the price per unit, and this enables the calculation of units bought with a reinvested dividend.

The upgraded `Transaction` class definition is listed below.

```matlab
classdef Transaction
    %Transaction defines a Transaction class to represent a transaction of
    %an investment assset, represented by the Asset class.
    %

    properties
        date % a MATLAB datetime object
        transactionType % 'buy', 'sell', 'short', 'dividend', 'split'
        quantity = 0;
        dividend = 0;
        split_ratio = 1;
        fee = 0; % Default value: zero
        price

    end % END: properties

    methods
        function obj = Transaction( varargin )
            %Transaction constructs an instance of the Transaction class.
            %
            % SYNTAX:
            %
            % newTrans = Transaction( Ttype, Tquantity, Tprice)
            %     Defines a new transaction with the current date/time in
            %     the date field. Valid transaction types are 'buy',
            %     'sell', and 'div-rnv' (dividend reinvestment).
            %
```

```
        % newTrans = Transaction( Tdate, Ttype, Tquantity, Tprice)
        %     Defines a new transaction with a specified date/time. The
        %     date/time object may be specified as MATLAB datetime
        %     object or as a 3- or 6-element date vector.
        %
        % newTrans = Transaction( Tdate, Ttype, Tquantity, Tprice, fee)
        %     Defines a new transaction with a specified date/time and
        %     a transaction fee.
        %

        % This switch-case control group defines the Transaction object
        % differently based on the
        switch nargin
            % The 3-input case assumes that the date is now
            case 3 % newTr = Transaction(Ttype, Tquantity, Tprice)
                obj.date = datetime('now');
                obj.transactionType = varargin{1};
                obj.quantity = varargin{2};
                obj.fee = 0;
                obj.price = varargin{3};

            % The 4-input case allows the transaction date to be
            % specified
            case 4 % newTr = Transaction(Tdate, Ttype, Tquantity, ...
                %              Tprice)

                switch class(varargin{1})
                    case 'datetime'
                        obj.date = varargin{1};
                    case 'double' % the date specifier is in vector form
                        % convert a date vector to a datetime object
                        obj.date = datetime(varargin{1});
                    otherwise
                        error('Invalid transaction date specification.')
                end

                obj.transactionType = varargin{2};
```

```
                obj.price = varargin{4};

                switch obj.transactionType
                    case 'div-rnv'
                        obj.dividend = varargin{3};
                        obj.quantity = obj.dividend/obj.price;
                    otherwise
                        obj.quantity = varargin{3};
                end


            case 5
                % newTr = Transaction( Tdate, Ttype, Tquantity, ...
                %           Tprice, Tfee )

                switch class(varargin{1})
                    case 'datetime'
                        obj.date = varargin{1};
                    case 'double' % the date specifier is in vector form
                        % convert a date vector to a datetime object
                        obj.date = datetime(varargin{1});
                    otherwise
                        error('Invalid transaction date specification.')
                end

                obj.transactionType = varargin{2};
                obj.quantity = varargin{3};
                obj.price = varargin{4};
                obj.fee = varargin{5};

            otherwise
                error('Invalid number of input arguments.')
        end

    end


    function listDetails(obj)
        DateString = [char(obj.date) ...
            blanks(20 - length(char(obj.date)) ) ];
        TypeString = [blanks(10-length(obj.transactionType)), ...
            obj.transactionType];
```

```matlab

            QtyString = [blanks(10 - length(num2str(obj.quantity))), ...
                num2str(obj.quantity)];

            RawPriceStr = num2str(obj.price, '%0.3g');
            PriceString = [' at ', blanks(10 - length(RawPriceStr)), ...
                RawPriceStr];

            DetailString = [DateString, TypeString, QtyString, PriceString];

            disp(DetailString)
        end

    end % END: methods
end
```

Listing 19: The `Transaction` *class, as enhanced to support* `Asset` *value calculations.*

Next, we list the upgraded `Asset` class definition with the new `calculateValue()` function:

```matlab
classdef Asset
    %Asset defines an Asset class to store information about a particular
    %investment asset (stock, mutual fund, currency, etc.).
    %
    % By E.P. Blair
    % Baylor University

    properties
        name = 'X-ray Yankee Zulu'
        symbol = 'XYZ'
        units = 'shares'
        transactionList
    end

    methods
        function obj = Asset(AName, ASymbol, AUnits)
            %Asset Constructs an instance (obj) of the Asset class
            %   Here, the required syntax is
            %   >> myObj = Asset( nameStr, symbolStr, unitStr )
            %   where nameStr, symbolStr, and unitsStr are char strings
            %   specifying the asset name, symbol, and units ('shares',
            %   'USD', 'RMB', etc.).
            obj.name = AName;
```

```
            obj.symbol = ASymbol;
            obj.units = AUnits;
        end

        function obj = addTransaction(obj, newTransaction)
            % addTransaction() adds a transaction newTransaction to the
            % transactionList property of obj

            % add the transaction on the end of the list
            obj.transactionList = [obj.transactionList newTransaction];

            % if the length of the list is greater than 1, the list may
            % require sorting

            if length(obj.transactionList) > 1
                % Create a list of transaction dates by iterating through
                % all transactions and adding dates to dateList
                dateList = []; % empty date list
                for TransIdx = 1:length(obj.transactionList)
                    % append the date of obj.transactionList(TransIdx) to
                    % dateList (unsorted)
                    dateList = [dateList ...
                        obj.transactionList(TransIdx).date];
                end

                % sort the transaction list

                % obtain an index of sorted transaction dates
                %   The sort function returns the sorted list along with
                %   the indices of sorted lists within the unsorted list
                %   The indices of the dateList, sortIndex, will be used to
                %   sort the list of transactions.
                [~, sortIndex] = sort(dateList);

                % reorder the unsorted transactions and store in
                % obj.transactionList
                obj.transactionList = obj.transactionList(sortIndex);
            end % END: if length(obj.transactionList) > 1

```

```matlab
        end

        function obj = listTransactions(obj)
            numTrans = length(obj.transactionList);

            if numTrans > 0
                disp(['Transactions for ', obj.name, ' (', obj.symbol, ...
                    '):'])
                for transIndex = 1:numTrans
                    obj.transactionList(transIndex).listDetails;
                end

            else
                disp(['Asset ', obj.name, ' (', obj.symbol, ...
                    ') has no transactions.'])
            end

        end

        function varargout = calculateValue(obj)
            % Asset/calculateValue performs an analysis on the list of
            % transactions to calculate asset holdings and their value at
            % the time of the last transaction.
            %
            % Syntax:
            %
            % Value = myAsset.calculateValue returns the value of holdings
            %     at the time of the last transaction.
            %
            % [Value, Units] = myAsset.calculateValue additionally returns
            %     the total number of units held.
            %
            % [Value, Units, CostBasis] = myAsset.calculateValue returns
            %     the investor's cost basis.
            %

            Value = 0; % value of holdings
            TotalUnits = 0; % number of units held
            CostBasis = 0; % cost basis of investment
            if ~isempty(obj.transactionList)
                numTrans = length(obj.transactionList);

```

```
% Units: storage vector for units owned as a fcn. of time
Units = zeros(1, numTrans);
Cost = zeros(1, numTrans);
Value = zeros(1, numTrans);

% Iterate through all transitions
for transIdx = 1:numTrans
    % extract a single transition
    tempTrans = obj.transactionList(transIdx);

    if transIdx == 1
        Units(1) = tempTrans.quantity;
        Cost(transIdx) = tempTrans.price ...
            * tempTrans.quantity ...
            + tempTrans.fee;

    else


        switch tempTrans.transactionType
            case 'buy'
                Units(transIdx) = Units(transIdx-1) ...
                    + tempTrans.quantity;

                Cost(transIdx) = tempTrans.price ...
                    * tempTrans.quantity ...
                    + tempTrans.fee;

            case 'sell'
                Units(transIdx) = Units(transIdx-1) ...
                    - tempTrans.quantity;
                Cost(transIdx) = -tempTrans.price ...
                    * tempTrans.quantity ...
                    + tempTrans.fee;

            case 'split'
                Units(transIdx) = Units(transIdx-1) ...
                    * tempTrans.split_ratio;

            case 'div-rnv'
                Units(transIdx) = Units(transIdx-1) ...
                    + tempTrans.dividend/tempTrans.
```

```
                                       price;
147
148                         end
149
150
151                     end
152
153                 end % END: for transIdx = 1:numTrans
154
155                 CostBasis = sum(Cost);
156                 Qty = Units(end);
157                 Value = Qty*tempTrans.price; %
158                 LastPrice = tempTrans.price;
159
160             end
161
162
163             switch nargout
164                 case 1
165                     varargout{1} = Value;
166                 case 2
167                     varargout{1} = Value;
168                     varargout{2} = Qty;
169                 case 3
170                     varargout{1} = Value;
171                     varargout{2} = Qty;
172                     varargout{3} = CostBasis;
173                 case 4
174                     varargout{1} = Value;
175                     varargout{2} = Qty;
176                     varargout{3} = CostBasis;
177                     varargout{4} = LastPrice;
178
179                 otherwise
180                     error('Invalid number of output arguments.')
181             end
182
183         end
184
185     end % END: methods
186 end
```

Listing 20: The `Asset` class with a new `calculateValue()` method.

The new `Asset/calculateValue()` method is listed in lines 82-174 of Listing 20. The function header uses `varargout` (a variable-length set of output arguments) to allow the user flexibility in outputs. The help documentation comments provide information about the syntax; and a `switch-case` control (lines 160-172) manages the outputs depending on `nargout`, the number of outputs in the particular function invocation.

Finally, we list a new testbed script, `testbedAssetv02.m` (see Listing 21). This defines an `Asset` object `newAsset`. It adds several transactions to `newAsset` and lists them using the `Asset` method `listTransactions()`. Finally, the script invokes the `Asset/calculateValue()` method and lists data calculated for this asset. Here, one `disp()` command was used, and the command `char(10)` embeds character ten (the MATLAB code for a newline character) in the string and breaks the string up for readability.

```
% testbedAssetv02.m

% create an new Asset object with no transactions
newAsset = Asset('X-ray Yankee Zulu, Inc.', 'XYZ', 'shares');

% add a buy transaction with the current date
newAsset = newAsset.addTransaction( Transaction('sell', 40, 24.03) );

% list transaction data after the first addition
newAsset.listTransactions;

% add a transaction with an earlier date
newAsset = newAsset.addTransaction( Transaction([2017, 1, 1], ...
    'buy', 25, 22.97) );

% add a transaction with an earlier date
newAsset = newAsset.addTransaction( Transaction([2017, 3, 18], ...
    'div-rnv', 40, 23.58) );

% add a transaction with an earlier date
newAsset = newAsset.addTransaction( Transaction([2017, 5, 24], ...
    'buy', 100, 25.17) );

% list transaction data after the first addition
newAsset.listTransactions;

[Value, Holdings, CostBasis] = newAsset.calculateValue;
Gain = Value - CostBasis;

disp([char(10), 'Asset            : ', newAsset.name, ' (', newAsset.symbol, ...
    ')', char(10), 'Quantity         : ', num2str(Holdings), ...
    char(10), 'Value            : $', num2str(Value), ...
    char(10), 'Cost Basis       : $', num2str(CostBasis), ...
    char(10), 'Unrealized Gains: $', num2str(Gain), ...
    ' or ', num2str(100*Gain/CostBasis, '%0.4g'), '%'])
```

Listing 21: A testbed function for `Asset` value calculations.

Running the testbed script of Listing 21 yields the output of Listing 22.

```
>> testbedAssetv02
```

```
Transactions for X-ray Yankee Zulu, Inc. (XYZ):
29-Dec-2017 20:46:36       sell         40 at         24
Transactions for X-ray Yankee Zulu, Inc. (XYZ):
01-Jan-2017                 buy         25 at         23
18-Mar-2017              div-rnv     1.6964 at       23.6
24-May-2017                 buy        100 at       25.2
29-Dec-2017 20:46:36       sell         40 at         24

Asset            : X-ray Yankee Zulu, Inc. (XYZ)
Quantity         : 86.6964
Value            : $2083.3134
Cost Basis       : $2130.05
Unrealized Gains: $-46.7366 or -2.194%
```

Listing 22: Output for the testbed function of Listing 21.

Some additional formatting may be desired for dollar and percentage amounts.

4.5 A `Portfolio` Class: a Container Class for `Asset` Objects

Here, we will define a `Portfolio` class that serves as a container class for `Asset` objects. Actually, we already created the `Asset` class as a container for `Transaction` objects. The `Portfolio` class will be built with an `addAsset()` method and a `calculateValue()` method. The `Portfolio` `calculateValue()` method will hierarchically calculate its own value by invoking the `Asset` class `calculateValue()` method for each `Asset` object held in the portfolio.

```
classdef Portfolio
    %PORTFOLIO defines a container class Portfolio for objects of
       type
    %   Asset. A Portfolio object calculates its own value by
       calling each
    %   contained Asset object to evaluate and return its individual
        value.
    %

    properties
        name
        assetList
    end

    methods
        function obj = Portfolio(varargin)
            %Portfolio constructs an Portfolio object.
            %
            %   SYNTAX:
            %
            %   myPortfolio = Portfolio creates an empty portfolio.
            %
            %   myPortfolio = Portfolio( portfolioName, assetArray )
            %   Detailed explanation goes here
```

```
            switch nargin
                case 0
                    obj.name = 'Default Portfolio';
                    obj.assetList = [];

                case 2
                    obj.name = varargin{1};
                    if strcmp(class(varargin{2}), 'Asset') % input checking
                        obj.assetList = varargin{2};
                    else
                        error('Non-Asset object specified for asset list.')
                    end

            end
        end % END: Portfolio constructor

        function obj = addAssets(obj, additionalAssets)
            % Portfolio/addAsset adds new Asset objects to the assetList
            %   property of a Portfolio object.
            %
            % SYNTAX:
            %
            % myPortfolio = myPortfolio.addAsset( additionalAssets )
            %
            if strcmp(class(additionalAssets), 'Asset') % input checking
                obj.assetList = [obj.assetList additionalAssets];
            else
                error('Non-Asset object specified for asset list.')
            end
        end % END: Portfolio/addAssets()

        function varargout = calculateValue(obj)

            PortfolioValue = 0;
            PortfolioData = [];

            if ~isempty(obj.assetList)
                numAssets = length(obj.assetList);
                Symbol = cell(numAssets, 1);
                Value = zeros(numAssets, 1); % storage vector
                Holdings = Value; % storage vector
                CostBasis = Value; % storage vector
                UnitPrice = Value;
```

```
                for AssetIdx = 1:numAssets
                    tempAsset = obj.assetList(AssetIdx);
                    [Value(AssetIdx), Holdings(AssetIdx), ...
                        CostBasis(AssetIdx), ...
                        UnitPrice(AssetIdx)] = tempAsset.
                           calculateValue;
                    Symbol{AssetIdx} = tempAsset.symbol;
                end
                UnrealizedGains = Value - CostBasis;

                PortfolioValue = sum(Value);
                PortfolioData = table(Symbol, Value, Holdings, ...
                    UnitPrice, CostBasis, UnrealizedGains);
            end

            switch nargout
                case 1
                    varargout{1} = PortfolioValue;
                case 2
                    varargout{1} = PortfolioValue;
                    varargout{2} = PortfolioData;
                otherwise
                    error('Invalid number of outputs specified.')
            end

        end % END: Portfolio/calculateValue()
    end
end
```

Listing 23: The class definition for `Portfolio`, a container class for objects of class `Asset`.

The key property of `Portfolio` in Listing 23 is the `assetList` property. This property will store a horizontal concatenation of `Asset` objects. The concatenation is seen in the `Portfolio` class `addAssets()` method. Finally, the `Portfolio` class `calculateValue()` method invokes the `Asset` class `calculateValue()` method on `Asset` objects contained by the `Portfolio` object. For each `Asset` object, `Portfolio` `calculateValue()` method saves information about holdings at the last transaction, including: value, total holdings, unit price, symbol, and the price per unit. This data then is combined in a MATLAB `table` object.

Next, I list a testbed function, `testbedPortfolio.m` in Listing 24. Here, two `Asset` objects are created, and several `Transaction` objects are added to each. Then, the two `Asset` objects are added to a new `Portfolio` object using the `Portfolio` method `addAssets()`. Finally, the `Portfolio` method `calculateValue()` is invoked on the new `Portfolio` object, and the returned data is printed using the `disp()` function.

```
% testbedPortfolio.m

% Define firstAsset and add some transactions
firstAsset = Asset('X-ray Yankee Zulu', 'XYZ', 'shares');
firstAsset = firstAsset.addTransaction( Transaction([2015, 10, 1],
```

```
    ...
    'buy', 100, 26.75) );
firstAsset = firstAsset.addTransaction( Transaction([2016, 5, 1], ...
    'sell', 25, 32.18) );
firstAsset = firstAsset.addTransaction( Transaction([2016, 12, 28], ...
    'div-rnv', 75.29, 33.42) );
firstAsset = firstAsset.addTransaction( Transaction([2017, 4, 1], ...
    'buy', 30, 32.18) );
firstAsset = firstAsset.addTransaction( Transaction([2017, 12, 27], ...
    'div-rnv', 82.15, 36.25) );

secondAsset = Asset('Quebec Romeo Sierra', 'QRS', 'shares');
secondAsset = secondAsset.addTransaction( Transaction([2014, 3, 8], ...
    'buy', 50, 13.28) );
secondAsset = secondAsset.addTransaction( Transaction([2014, 12, 29], ...
    'div-rnv', 42.69, 16.24) );
secondAsset = secondAsset.addTransaction( Transaction([2015, 6, 24], ...
    'buy', 20, 17.01) );
secondAsset = secondAsset.addTransaction( Transaction([2015, 12, 28], ...
    'div-rnv', 53.12, 17.79) );
secondAsset = secondAsset.addTransaction( Transaction([2016, 8, 13], ...
    'buy', 50, 18.24) );
secondAsset = secondAsset.addTransaction( Transaction([2016, 12, 27], ...
    'div-rnv', 62.24, 19.13) );


newPortfolio = Portfolio; % create an empty Portfolio object

% add the newly-created assets as
newPortfolio = newPortfolio.addAssets([firstAsset, secondAsset]);

[PortfolioValue, PortfolioData] = newPortfolio.calculateValue;
TotalValue = sum(PortfolioData.Value);
TotalCostBasis = sum(PortfolioData.CostBasis);
TotalGains = sum(PortfolioData.UnrealizedGains);
disp(['Total portfolio value: $', num2str(TotalValue), char(10), ...
    'Cost basis           : $', num2str(TotalCostBasis), char(10), ...
```

```
    'Unrealized gains     : $', num2str(TotalGains), char(10)]);
```

Listing 24: The class definition for `Portfolio`, a container class for objects of class `Asset`.

The output for Listing 24 is shown below in Listing 25:

```
>> testbedPortfolio
Total portfolio value: $6435.3136
Cost basis           : $4752.1
Unrealized gains     : $1683.2136
```

Listing 25: Output generated by `testbedPortfolio.m`, listed in Listing 24.

5 Extended Example: Class Infrastructure for a Multi-player Game

In this example, we develop some classes that may support a multi-player game, and we apply some visualization concepts in drawing a rudimentary game arena with avatars representing players. Therefore, we begin by defining an `Avatar` class. The example will continue with visualization of multiple avatars within a single `axes` object.

5.1 Defining an `Avatar` Class

We show a rudimentary `Avatar` class definition in Listing 26. Properties typical of a player's digital representation in a role-playing video game are present. The constructor method allows several different invocation syntaxes. Also, we have defined a `move()` method, which is used to change the `Avatar` object's position on the battlefield.

```
classdef Avatar
    %AVATAR objects represent a player in a multi-player role-playing game.
    %  The AVATAR class defines properties typical of avatars in RPGs.
    %
    % By E.P. Blair
    % Baylor University

    properties
        Name = 'Unidentified Player';
        Class % {'mage', 'warrior', 'thief', 'jedi', 'sith', 'none'}
        Level = 1; % a numerical rank for a player
        Position = [0 0]; % (x, y) double for
        XP = 0; % a numerical property for accumulating experience points
        HealthPointsMax = 100; % player's maximum health points (HP)
        Vitality = 1 % player's actual vitality (fraction of maximum vitality)
                  % HP = Vitality * HealthPointsMax
        Attack % numerical rating for offensive capabilities
        Defense % numerical rating for densive capabilities
        WeaponList % list of player's a offensive equipment
        ArmorList % list of player's a defensive equipment
        EquipmentList % list of player's equipment items
    end % END: properties

    methods
        function obj = Avatar(varargin)
            %Avatar instantiates an AVATAR object
            %   SYNTAX
            %
            %   newAvatar = Avatar creates a default AVATAR object
            %
```

```
            %   newAvatar = Avatar(Username) creates a default AVATAR
            %           object and specifies Username as the name.
            %
            %   newAvatar = Avatar(Username, PlayerClass) creates a default
            %            AVATAR object with name Username and class
            %            PlayerClass.
            %

            switch nargin
                case 0
                    obj.Class = 'none';
                case 1
                    obj.Name = varargin{1};
                    obj.Class = 'none';
                case 2
                    obj.Name = varargin{1};
                    obj.Class = varargin{2};
            end
        end

        function obj = move(obj, dispVect)
            % myPlayer = myPlayer.move( [dX dY] ) displaces the avatar on
            %   the battlefield.
            obj.Position = obj.Position + dispVect;
        end

        function disp(obj)
            % disp(someAvatar) is the display function for the AVATAR
            % class.
            disp([obj.Name, ' (Level ', num2str(obj.Level), ' ', ...
                obj.Class, ') is at (', ...
                num2str(obj.Position(1)) , ', ', ...
                num2str(obj.Position(2)), ').']);
        end

    end % END: methods
end
```

Listing 26: An initial class definition file for the `Avatar` *class.*

5.1.1 Overriding the `disp()` Function

We also have defined a `disp()` function. Each bulit-in MATLAB class has its own particular `disp` function, which is invoked to display information about an object in question when an unsuppressed

MATLAB calculation yields an object of that class. For a user-defined class, if no `disp()` method is defined, MATLAB displays that object in a manner similar to the display of a `struct`, giving a listing of all of the fields pertinent to that particular class. Here, we have a customized `disp()` method that is written to display the player's name, level and class, as well the battlefield position of the player's `Avatar` object. When a method `someMethod()` is defined for a class, but `someMethod()` already exists for other classes, we say that we have **overridden** the `someMethod()` method. Here, we have overridden the `disp` method to define a customized format for the display of information for objects of class `Avatar`.

As an example, see Listing 27, where we test the new `Avatar` class definition in the Command Window. When the `Avatar` constructor is invoked (and not suppressed), the result is an object of class `Avatar`, so the `Avatar` class `disp()` method is invoked to display information about the resulting `Avatar` object:

```
>> newAvatar = Avatar('Sargon', 'warrior')

newAvatar =

Sargon (Level 1 warrior) is at (0, 0).
```

Listing 27: Command Window input and output to test the ***`Avatar`*** *class constructor and overridden* ***`disp()`*** *method.*

5.1.2 Testing the `move()` Method

Next, we test the `move()` method in the Command Window:

```
>> newAvatar = newAvatar.move([27, -126])

newAvatar =

Sargon (Level 1 warrior) is at (27, -126).
```

Listing 28: The ***`Avatar`*** *class* ***`move()`*** *method worked as desired in a Command Window test.*

5.2 Graphical Visualization for the `Avatar` Class

This is where writing the `Avatar` class gets fun and challenging. First, we will add a `draw()` method which draws a graphical representation of an `Avatar` object on an axes. This calls for adding a property to the `Asset` class which stores a handle to the drawing. When the `draw()` method is invoked, it can then check to see if the `Avatar` object already has a drawing; if so, we need not draw it again. Then, we will upgrade the `move()` method so that it not only changes the `Position` property, but also updates the `Avatar` object's drawing, as applicable.

5.3 The `Avatar` Class `draw()` Method

Listing 29 shows the new snippets of the `Avatar` class definition file.

```
classdef Avatar
    %AVATAR objects represent a player in a multi-player role-
        playing game.
```

```
    %   The AVATAR class defines properties typical of avatars in RPGs.
    %
    % By E.P. Blair
    % Baylor University

    properties

        <-- snip -->

        % Graphics-related properties
        Drawing % A struct of handles to the drawing components

    end % END: properties

    methods

        <-- snip -->

        function obj = draw(obj, varargin)

            % Default values
            TargetAxes = [];
            % Override default values: parse varargin for property-value
            %   pairs

            args = varargin;
            while length(args) >= 2
                prop = args{1};
                val = args{2};
                args = args(3:end);

                switch prop
                    case 'Axes'
                        TargetAxes = val;
                    otherwise
                        error(['''', prop, ''' is an invalid ', ...
                            'property specifier.'])
                end % END switch prop
            end % END while length(args) >= 2

            if isempty(obj.Drawing)

                % DRAW THE SQUARE (MAIN BODY)
                % Calculate relative points of corners
                x_rel = [-5 -5 5 5];
                y_rel = [-5 5 5 -5];
```

```
        % Calculate absolute points of corners
        x = obj.Position(1) + x_rel;
        y = obj.Position(2) + y_rel;
        % visualization
        Drawing.Body = patch(x, y, [1 1 1], ...
            'EdgeColor', [0 0 0], 'LineWidth', 2);

        NameText = text(obj.Position(1), obj.Position(2), ...
            obj.Name, 'FontName', 'Times', 'FontSize', 18, ...
            'HorizontalAlignment', 'center', ...
            'VerticalAlignment', 'bottom');

        % PLAYER INFO STRING
        % 'LVL. X C' (X = Level, C = Class)
        PlayerInfoStr = ['LVL. ', num2str(obj.Level), ' ', ...
            upper(obj.Class(1:3))];
        xPInfoStr = obj.Position(1);
        yPInfoStr = obj.Position(2) + 4;
        PlayerInfoText = text(xPInfoStr, yPInfoStr, PlayerInfoStr, ...
            'FontName', 'Times', 'FontSize', 14, ...
            'HorizontalAlignment', 'center', ...
            'VerticalAlignment', 'middle');


        % HEALTH BAR
        % Full health will span [-4, 4] (relative)
        % No health is a point at -4 (relative
        % Color will transition from [0 0.5 0] (green, full)
        % to [1 0 0] (red, empty)
        cHealthLine = [ 1-obj.Vitality, 0.75*obj.Vitality, 0];
        xHealthLine = obj.Position(1) + [-4, -4+8*obj.Vitality];
        yHealthLine = obj.Position(2) - [2 2];
        HealthLine = line( xHealthLine, yHealthLine, ...
            'Color', cHealthLine, ...
            'LineWidth', 5);

        % HEALTH STATUS STRING (HealthStr)
        % 'HP: XX/MAX'
        HealthStr = ['HP: ', ...
            num2str(round(obj.Vitality*obj.HealthPointsMax)), ...
            '/', num2str(obj.HealthPointsMax)];
```

```
 90                     xHealthStr = obj.Position(1) + 4;
 91                     yHealthStr = obj.Position(2) - 3.5;
 92                     HealthText = text(xHealthStr, yHealthStr, HealthStr,
                            ...
 93                         'FontName', 'Times', 'FontSize', 14, ...
 94                         'HorizontalAlignment', 'right', ...
 95                         'VerticalAlignment', 'middle');
 96 
 97                     % Populate the Drawing struct
 98                     Drawing.PlayerInfoText = PlayerInfoText;
 99                     Drawing.HealthText = HealthText;
100                     Drawing.Health = HealthLine;
101 
102                     % store Drawing in obj.Drawing
103                     obj.Drawing = Drawing;
104 
105                 end
106             end
107 
108     end % END: methods
109 end
```

Listing 29: A `draw()` method was added to the `Avatar` class.

The `draw()` method provides some nice features here:

- `draw()` supports optional property-value pairs using `varargin`. A block is reserved for default values (lines 22-23), which then can be optionally overridden by using the property-value pairs (lines 28-41).
- Line 43 is used to evaluate whether drawing is necessary. Drawing only commences if the `obj.Drawing` object is empty, which is the case for any newly-created `Avatar` object (see the newly-added `Drawing` property on line 13).
- Drawing begins with a square patch of side length 10, centered at the `Avatar` object's position (lines 45-54).
- `draw()` writes the character's name in the center of the square patch. See lines 56-59.
- `draw()` provides a player information string, with character level and the first three letters of the character class. See lines 61-70.
- A health bar is shown below the character name. Its length decreases as the character's vitality decreases from full health (`Vitality = 1`) to no health (`Vitality = 0`). Additionally, the health bar will turn red as `Vitality` approaches 0. See lines 73-83.
- A health string is shown at the bottom, including the current number of health points and the maximum number of health points for the player (lines 85-96).

Thus, the result of the testbed function `testbedAvatar.m` (Listing 30) is the MATLAB `figure` shown in Fig. 6. At this point, the `draw()` method seems to work as well as designed. Next steps include:

- Upgrading the `move()` method to update the graphics objects stored within the `Drawing` property.
- Adding functions such an `updateStatus()` method, which and update the data represented in the drawing. This method also could be called by other methods, such as `injure()` and `heal()`, which adjust the player's `Vitality` property, along with updating the drawing data.

```
% testbedAvatar.m
close all;
clear all;

Player1 = Avatar('Sargon', 'warrior') % create Player1

Player1 = Player1.move([-3, 13]) % displace Player1

Player1 = Player1.draw; % execute draw()

xlim([-20, 20]); % adjust the x limits beyond the square patch
axis equal % make the x- and y-scales equal
```

Listing 30: A testbed function tests the new `draw()` method.

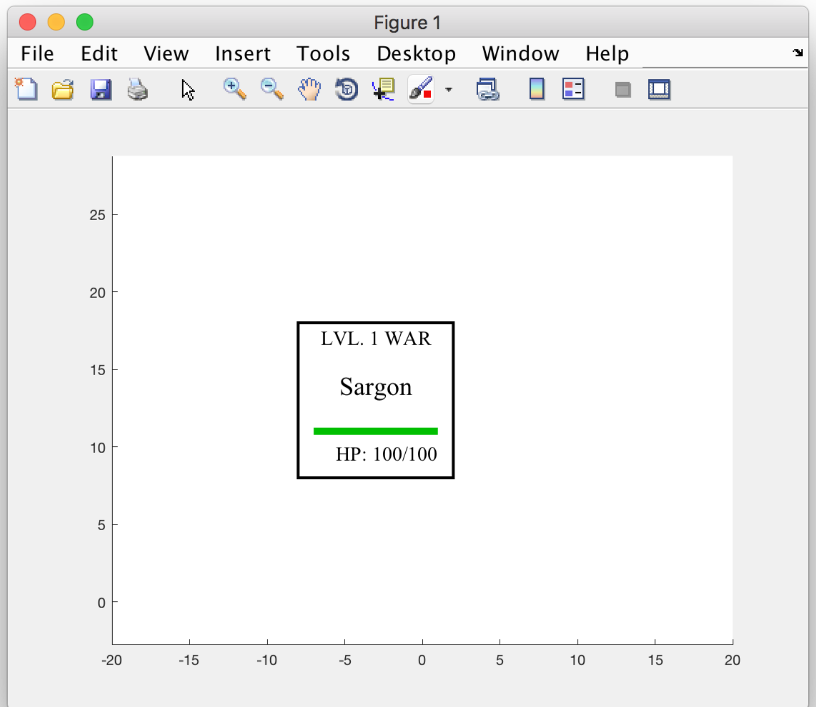


Figure 6: A rudimentary graphical representation for an `Avatar` *object. Here,* `Player1` *has the name* `'Sargon'` *and is drawn to indicate level 1 status as a warrior with full health (100/100 HP).*

A Control Statements

A.1 if-elseif-else-end

Perhaps the simplest control statement is the achieved using `if-elseif-else-end`. The simplest version of this is an `if-end` statement, with the following typical syntax in a script or function:

```
if ctrl_expr
  statements
end
```

Here, the control begins with `if ctrl_expr` and ends with the key word `end`. The statements are executed if the real part of the control expression `ctrl_expr` has all non-zero elements.

Most typically, `ctrl_expr` is an expression which evaluates to a `logical` or a `double` value. An example of this is

```
% basic_if_control.m

a = true
if a
    disp('''a'' is true.')
end
```

The output of `basic_if_control.m` is

```
>> basic_if_control

a =

  logical

   1

'a' is true.
```

Notice that `a` is a logical value, and it results in the execution of the block within the `if-end` construct.

Optionally, the `else` key word adds a section which is executed if the preceding sections of the `if` control is not executed. The `if-else-end` syntax is demonstrated as follows.

As an example of this, consider the function `sign_fun()`:

```
function sign_fun(x)
% sign_fun(x) prints a message about the negativity of x.
%
% By E.P. Blair
% Baylor University

if x < 0
    disp('x is negative.')
else
    disp('x is non-negative.')
end
```

This function was invoked on the command line with several test inputs:

```
>> sign_fun(-5)
x is negative.
>> sign_fun(-1.25)
x is negative.
>> sign_fun(0)
x is non-negative.
>> sign_fun(1)
x is non-negative.
>>
```

The `if-else-end` syntax allows for two different outcomes. Optional `elseif` blocks may be added to support additional possible outcomes. Consider, for example, this improved version of `sign_fun()`:

```
function sign_fun(x)
% sign_fun(x) prints a message about the sign of x.
%
% By E.P. Blair
% Baylor University

if x < 0
    disp('x is negative.')
elseif x > 0
    disp('x is positive.')
else
    disp('x is zero.')
end
```

The command line inputs and outputs for several tests are shown below.

```
>> sign_fun(-2)
x is negative.
>> sign_fun(3)
x is positive.
>> sign_fun(0)
x is zero.
```

Multiple `elseif` blocks may be added, as in the following function, which assigns a letter grade given a percentage score:

```
function lg = letter_grade(percentScore)
% letter_grade maps a percentage score to a letter grade.

if percentScore < 60
    lg = 'F';
elseif percentScore < 70
    lg = 'D';
elseif percentScore < 80
    lg = 'C';
```

```
elseif percentScore < 90
    lg = 'B';
else
    lg = 'A';
end
```

A.2 `for` Loops

A.3 `switch-case` Controls

A `switch-case` is useful when a finite, discrete set of cases may occur. The syntax for a `switch-case` control is as follows:

```
switch expr
  case expr_1
    statement_group_1
  case expr_2
    statement_group_2
  ...
  otherwise
    statement_group_otherwise
end
```

Here, the controlling expression evaluates to either a `char` string or an integer. If `expr` matches `expr_1`, then only `statement_group_1` executes; if `expr` matches `expr_2`, then only `statement_group_2` executes. The `otherwise` key word defines another block of statements that executes if `expr` does not match any of the expressions following a `case` key word. `Asset/calculateValue()` avoids the calculation if the `Asset` object's `transactionList` property is empty by using an `if ~isempty(obj.transactionList)` block. Thus, the block executes if `obj.transactionList` is not empty. Inside this block, a `for` loop iterates through each transaction and calculates/records the number of units transacted. `Asset/calculateValue()` uses a the price-per-unit data to determine the transaction cost, the total value, and the cost basis for the asset holdings.