

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	3
1.1	Object-oriented Programming	3
1.2	Data Types	3
1.2.1	Elementary Data Types	3
1.2.2	Arrays	3
1.2.3	Structures	4
1.3	Functions	5
1.4	Pre-defined Functions	5
1.4.1	Mathematical Functions	5
1.4.2	The <code>class()</code> Function	5
1.4.3	The <code>length()</code> Function	6
1.4.4	The <code>disp()</code> Function	8
1.4.5	The <code>num2str()</code> Function	8
1.4.6	Obtaining Time Intervals using <code>tic</code> and <code>toc</code>	8
1.5	User-defined Functions	9
1.5.1	The <code>varargin</code> Keyword	10
1.5.2	The <code>nargin</code> Function	11
1.6	Classes	12
1.6.1	Beyond Structures	12
1.6.2	Motivation for Classes	12
1.6.3	The Advantages of Classes	13
2	Built-in Classes	14
2.1	The <code>datetime</code> Class	14
2.1.1	Creating a New <code>datetime</code> Object	14
2.1.2	Properties	14
2.1.3	Method Functions	15
2.2	MATLAB Graphics Classes	16
2.2.1	The <code>Figure</code> Class	16
2.2.2	The <code>Axes</code> Class	17
2.2.3	The <code>Line</code> Class	20
2.2.4	The <code>Patch</code> Class	22
2.2.5	Animations using MATLAB Handle Graphics	25

3	Defining Classes in MATLAB	26
3.1	The Class-definition File	26
3.1.1	Modifying Objects	28
4	Extended Example: Developing Software to Track Investments	29
4.1	Developing an Asset Class	29
4.2	Developing a Transaction Class	30
4.3	Adding Transactions to an Asset Object	34
4.4	Calculating the Value of Holdings in an Asset	41
4.5	A Portfolio Class: a Container Class for Asset Objects	51
5	Extended Example: Class Infrastructure for a Multi-player Game	56
5.1	Defining an Avatar Class	56
5.1.1	Overriding the <code>disp()</code> Function	57
5.1.2	Testing the <code>move()</code> Method	58
5.2	Graphical Visualization for the Avatar Class	58
5.3	The Avatar Class <code>draw()</code> Method	58
A	Control Statements	64
A.1	<code>if-elseif-else-end</code>	64
A.2	<code>for</code> Loops	66
A.3	<code>switch-case</code> Controls	66

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 **doubles**, 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 **chars**, 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 `cells` may be elements of the same `cell`.

1.2.3 Structures

MATLAB allows the formation of **data structures** (or simply “structures” for short; and `structs` in MATLAB). A structure is a composite variable comprised of multiple pieces of data. MATLAB `structs` are very flexible, as a `struct` may contain data of elementary data types, arrays, cell arrays, other `structs`, 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:

```
1 >> R = struct('a', 7, 'b', 'life')
2
```

```

3 R =
4
5 struct with fields:
6
7     a: 7
8     b: 'life'
9
10 >>

```

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.

```

1 >> a = 5;
2 >> class(a)
3
4 ans =
5
6     'double'
7
8 >> class(pi)

```

```

9
10 ans =
11
12     'double'
13
14 >> b = rand(3, 4);
15 >> class(b)
16
17 ans =
18
19     'double'
20
21 >> class('s')
22
23 ans =
24
25     'char'
26
27 >> class('cat')
28
29 ans =
30
31     'char'
32
33 >> friend1.name = 'Alice'; friend1.age = 23; class(friend1)
34
35 ans =
36
37     '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:

```

1 >> A = 1:7
2
3 A =
4
5     1     2     3     4     5     6     7
6
7 >> length(A)
8
9 ans =
10
11     7

```

```

12
13 >> B = (12:16) '
14
15 B =
16
17     12
18     13
19     14
20     15
21     16
22
23 >> length(B)
24
25 ans =
26
27     5

```

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

```

1 >> C = rand(3,4)
2
3 C =
4
5     0.6541     0.4505     0.9133     0.5383
6     0.6892     0.0838     0.1524     0.9961
7     0.7482     0.2290     0.8258     0.0782
8
9 >> length(C)
10
11 ans =
12
13     4
14
15 >> D = rand(4,3)
16
17 D =
18
19     0.4427     0.7749     0.3998
20     0.1067     0.8173     0.2599
21     0.9619     0.8687     0.8001
22     0.0046     0.0844     0.4314
23
24 >> length(D)
25
26 ans =
27
28     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:

```
1 >> some_str = 'hello';
2 >> disp(some_str)
3 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:

```
1 >> x = 9;
2 >> disp(['x = ', num2str(x)])
3 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:

```
1 >> tic; % like starting a stopwatch
2 pause(0.25); % insert a pause to simulate some commands run
3 time_elapsed = toc % ends the timer and reports the time elapsed
4
5 time_elapsed =
6
7     0.2555
8
9 >>
```

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:

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



```

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

```

The output would be something like this:

```

1 First interval: 0.257 s
2 Second interval: 0.755 s
3 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:

```

1 function [out1, out2] = functionName( in1, in2, in3 )
2     statements
3 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:

```
1 function f = f_function(x, y, z)
2 %f_function calculates f(x, y, z), where
3 %   f(x,y,z) = x + y^2 + z^3
4 %
5 % By E.P. Blair
6 % Baylor University
7 %
8 f = x + y^2 + z^3;
9 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.

```

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

```

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

```

1 [a, b] = functionName( x, y, z );
2
3 [c, d] = functionName( x, y, z, e );
4
5 [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:

```

1 x = linspace(0, 1, 101);
2 y = sin(2*pi*x);
3 figure;
4 plot(x, y); % generates a default plot
5 figure % new figure
6 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:

```

1 function y = calc_poly(x, a, b, c, varargin)
2 %CALC_POLY calculates the value of a quadratic, cubic or quartic
3 %    expressions.
4 %
5 % SYNTAX
6 % =====
7 %    y = calc_poly(x, a, b, c ) evaluates the value of the quadratic
8 %    expression a*x^2 + b*x + c
9 %
10 %    y = calc_poly(x, a, b, c, d ) evaluates the value of the cubic
11 %    expression a*x^3 + b*x^2 + c*x + d
12 %

```

```

13 % y = calc_poly(x, a, b, c, d, f ) evaluates the value of the
14 %      quartic expression a*x^4 + b*x^3 + c*x^2 + d*x + f
15 %
16 % By. E.P. Blair
17 % Baylor University
18 % 2021.04.17
19
20 switch nargin % number of input arguments
21     case 4 % quadratic
22         y = a*x.^2 + b*x + c;
23
24     case 5 % cubic
25         d = varargin{1};
26         y = a*x.^3 + b*x.^2 + c*x + d;
27
28     case 6 % quartic
29         d = varargin{1};
30         f = varargin{2};
31         y = a*x.^4 + b*x.^3 + c*x.^2 + d*x + f;
32
33     otherwise
34         error(['calc_poly: invalid number (', num2str(nargin), ...
35             ') of input arguments.'])
36
37 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:

```
1 >> x = datetime('now')
2
3 x =
4
5     datetime
6
7     17-Apr-2021 19:44:34
8
9 >> class(x)
10
11 ans =
12
13     '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:

```
1 >> properties(x)
2
3 Properties for class datetime:
4
5     Format
6     TimeZone
```

```

7      Year
8      Month
9      Day
10     Hour
11     Minute
12     Second
13
14 >> x.Year
15
16 ans =
17
18         2021
19
20 >> disp(['Month: ', num2str(x.Month)])
21 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:

```

1 >> bday = datetime(2016, 10, 5)
2
3 bday =
4
5     datetime
6
7     05-Oct-2016
8
9 >> age = between(bday, datetime('now'))
10
11 age =
12
13     calendarDuration
14
15     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

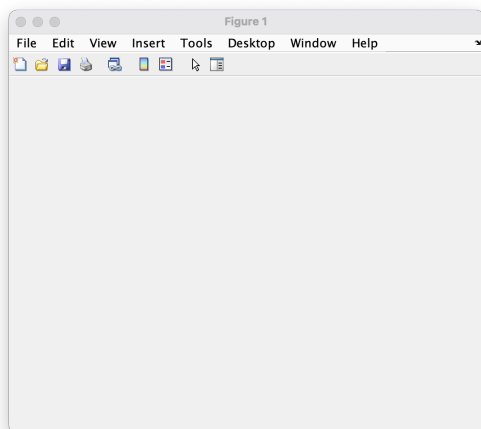
Consider the following Command Window session, which creates a new figure:

```
1 >> myFig = figure
2
3 myFig =
4
5     Figure (1) with properties:
6
7         Number: 1
8         Name: ''
9         Color: [0.9400 0.9400 0.9400]
10        Position: [617 599 560 420]
11        Units: 'pixels'
12
13    Show all properties
14
15 >> ishandle(myFig)
16
17 ans =
18
19     logical
20
21     1
22 >> m = 7; ishandle(m)
23
24 ans =
25
26     logical
27
28     0
29 >> myFig.Position
30
31 ans =
32
33     617     597     560     420
34
35 >>
```

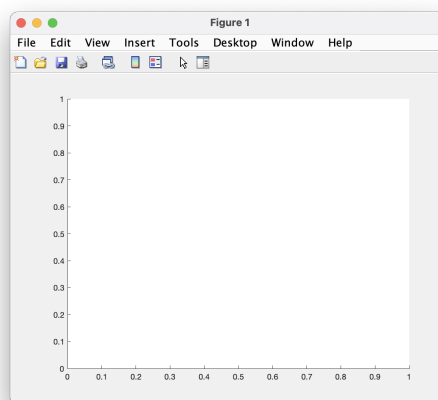
Listing 7: A new figure is created in the MATLAB Command Window.

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:

```
1 >> myAx = axes
2
3 myAx =
4
```

```

5  Axes with properties:
6
7      XLim: [0 1]
8      YLim: [0 1]
9      XScale: 'linear'
10     YScale: 'linear'
11     GridLineStyle: '-'
12     Position: [0.1300 0.1100 0.7750 0.8150]
13     Units: 'normalized'
14
15 Show all properties
16
17 >>

```

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:

```

1 >> myFig.Children % this points to an Axes object
2
3 ans =
4
5 Axes with properties:
6
7     XLim: [0 1]
8     YLim: [0 1]
9     XScale: 'linear'
10    YScale: 'linear'
11    GridLineStyle: '-'
12    Position: [0.1300 0.1100 0.7750 0.8150]
13    Units: 'normalized'
14
15 Show all properties
16
17 >>

```

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:

```

1 >> plot_a = subplot(2,1,1) % create an upper subplot
2
3 plot_a =
4
5 Axes with properties:
6
7     XLim: [0 1]

```

```

8         YLim: [0 1]
9         XScale: 'linear'
10        YScale: 'linear'
11    GridLineStyle: '-'
12        Position: [0.1300 0.5838 0.7750 0.3412]
13        Units: 'normalized'
14
15    Show all properties
16
17 >> plot_b = subplot(2,1,2) % create a lower subplot
18
19 plot_b =
20
21    Axes with properties:
22
23        XLim: [0 1]
24        YLim: [0 1]
25        XScale: 'linear'
26        YScale: 'linear'
27    GridLineStyle: '-'
28        Position: [0.1300 0.1100 0.7750 0.3412]
29        Units: 'normalized'
30
31    Show all properties
32
33 >> myFig.Children % now the figure has a two child Axes objects
34
35 ans =
36
37    2x1 Axes array:
38
39    Axes
40    Axes
41
42 >> myFig.Children(1) % points to the first Axes in the array
43
44 ans =
45
46    Axes with properties:
47
48        XLim: [0 1]
49        YLim: [0 1]
50        XScale: 'linear'
51        YScale: 'linear'
52    GridLineStyle: '-'
53        Position: [0.1300 0.1100 0.7750 0.3412]
54        Units: 'normalized'
55

```

```

56 Show all properties
57
58 >> myFig.Children(2) % points to the second Axes in the array
59
60 ans =
61
62 Axes with properties:
63
64         XLim: [0 1]
65         YLim: [0 1]
66         XScale: 'linear'
67         YScale: 'linear'
68         GridLineStyle: '-'
69         Position: [0.1300 0.5838 0.7750 0.3412]
70         Units: 'normalized'
71
72 Show all properties
73
74 >>

```

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:

```

1 >> x = linspace(-1, 1, 201);
2 % Plot two lines, with data_plot as a reference to the lines
3 data_plot = plot(x, sin(2*pi*x), x, cos(2*pi*x), 'LineWidth', 2)
4
5 data_plot =
6
7 2x1 Line array:
8
9 Line
10 Line
11
12 >> thisAx = gca; thisFig = gcf; thisAx
13
14 thisAx =
15
16 Axes with properties:
17
18         XLim: [-1 1]
19         YLim: [-1 1]
20         XScale: 'linear'

```

```

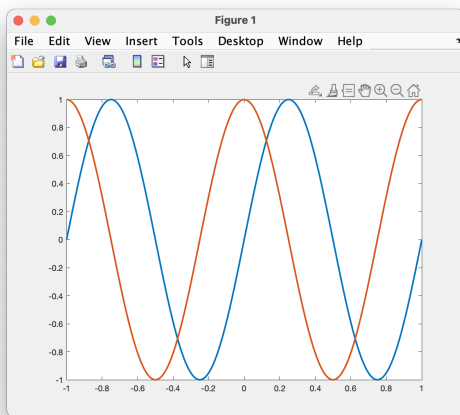
21         YScale: 'linear'
22     GridLineStyle: '-'
23     Position: [0.1300 0.1100 0.7750 0.8150]
24     Units: 'normalized'
25
26     Show all properties
27
28 >> thisAx.Children % the children of thisAx are Line objects
29
30 ans =
31
32     2x1 Line array:
33
34     Line
35     Line
36
37 >>

```

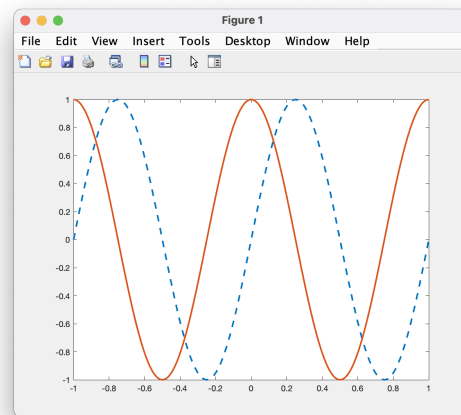
*Listing 11: Two data sets are plotted on one **Axes** object.*

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 wished 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 (RGB) 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`:

```

1 >> someLine.LineStyle = '--';
2 >> someLine.Marker = 'o';

```

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

```

1 >> 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 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:

```

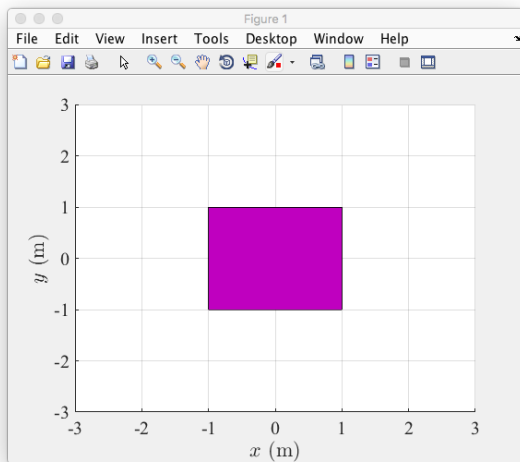
1 >> Dx = -2; Dy = 1.5;
2 >> set(newSquare, 'XData', newSquare.XData + Dx, 'YData', newSquare.
   YData + DY)

```

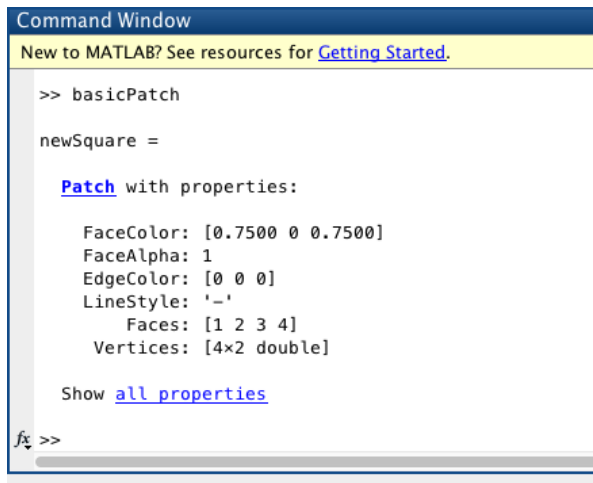
Listing 14: A Command Window modification to the `patch` defined in Listing 13. Line 1 is used to specify the x - and y -components of the displacement. The displacement is then applied to the original `XData` and `YData` properties, and the result of the addition is the value component of a property-value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

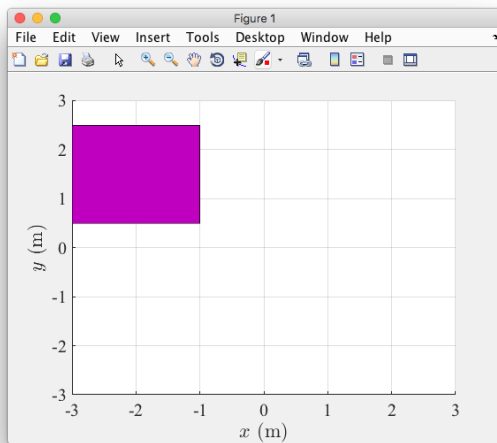


(a)

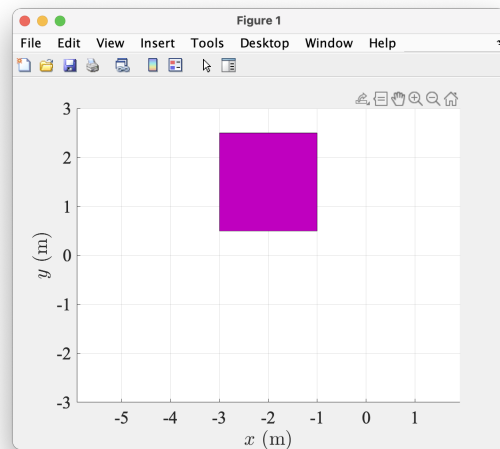


(b)

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



(a)



(b)

Figure 4: 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.


```
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 variable 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 file 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`.

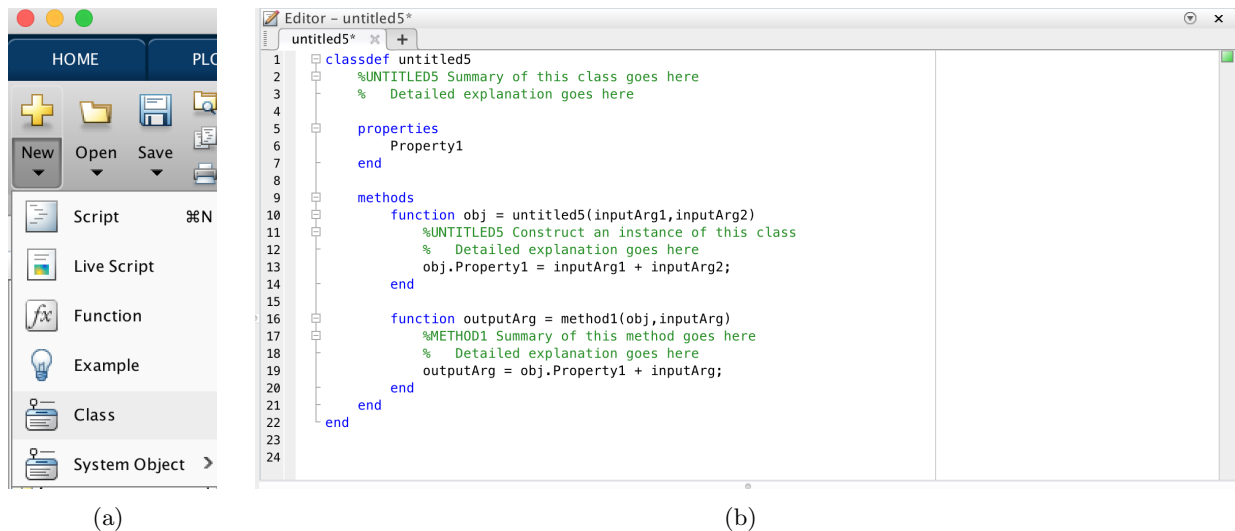


Figure 5: (a) The “New Document” button provides a “Class” option. (b) A new class template obtained in MATLAB 2017b.

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.

```
1 >> x = method1(myObj, 5)
2
3 x =
4
5      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:

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

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

```
1 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`:

```
1 % testbedAsset.m
2
3 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.

```
1 classdef Transaction
2     %Transaction defines a Transaction class to represent a
3     %transaction of
4     %an investment asset, represented by the Asset class.
5     %
6     properties
7         date % a MATLAB datetime object
8         transactionType % 'buy', 'sell', 'short', 'dividend', 'split'
9         quantity
10        fee
11        price
12
13    end % END: properties
14
15    methods
```

```

16 function obj = Transaction( varargin )
17     %Transaction constructs an instance of the Transaction
        class.
18
19     %
20     % SYNTAX:
21     % newTrans = Transaction( Ttype, Tquantity, Tprice)
22     %     Defines a new transaction with the current date/
        time in
23     %     the date field.
24     %
25     % newTrans = Transaction( Tdate, Ttype, Tquantity,
        Tprice)
26     %     Defines a new transaction with a specified date/
        time. The
27     %     date/time object may be specified as MATLAB
        datetime
28     %     object or as a 3- or 6-element date vector.
29     %
30     % newTrans = Transaction( Tdate, Ttype, Tquantity,
        Tprice, fee)
31     %     Defines a new transaction with a specified date/
        time and
32     %     a transaction fee.
33     %
34
35     % This switch-case control group defines the Transaction
        object
36     % differently based on the
37     switch nargin
38         % The 3-input case assumes that the date is now
39         case 3 % newTr = Transaction(Ttype, Tquantity,
            Tprice)
40             obj.date = datetime('now');
41             obj.transactionType = varargin{1};
42             obj.quantity = varargin{2};
43             obj.fee = 0;
44             obj.price = varargin{3};
45
46         % The 4-input case allows the transaction date to be
47         % specified
48         case 4 % newTr = Transaction(Tdate, Ttype, Tquantity
            , ...
49             %
            Tprice)
50
51             switch class(varargin{1})
52                 case 'datetime'
53                     obj.date = varargin{1};

```

```

54         case 'double' % the date specifier is in
55             vector form
56             % convert a date vector to a datetime
57             object
58             obj.date = datetime(varargin{1});
59         otherwise
60             error('Invalid transaction date
61                 specification.')
62         end
63
64         obj.transactionType = varargin{2};
65         obj.quantity = varargin{3};
66         obj.price = varargin{4};
67
68     case 5
69         % newTr = Transaction( Tdate, Ttype, Tquantity,
70             ...
71             Tprice, Tfee )
72
73         switch class(varargin{1})
74             case 'datetime'
75                 obj.date = varargin{1};
76             case 'double' % the date specifier is in
77                 vector form
78                 % convert a date vector to a datetime
79                 object
80                 obj.date = datetime(varargin{1});
81             otherwise
82                 error('Invalid transaction date
83                     specification.')
84             end
85
86         obj.transactionType = varargin{2};
87         obj.quantity = varargin{3};
88         obj.price = varargin{4};
89         obj.fee = varargin{5};
90
91     otherwise
92         error('Invalid number of input arguments.')
93     end
94
95 end
96
97 end % END: methods
98
99 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:

```

1 % testbedTransaction.m
2
3 % three-input constructor invocation
4 trans01 = Transaction( 'buy', 75, 71.90 )
5
6 % four-input constructor invocation with argument 1 as a date vector
7 trans02 = Transaction( [2017, 12, 1, 14, 30, 0], 'buy', 100, 7.19 )
8
9 % four-input constructor invocation with argument 1 as a datetime
   object
10 trans03 = Transaction( datetime('now'), 'sell', 25, 52 )
11
12 % five-input constructor to specify a transaction fee
13 trans04 = Transaction( datetime('now'), 'sell', 25, 52, 7 )

```

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

```

1 >> testbedTransaction
2
3 trans01 =
4
5     Transaction with properties:
6
7         date: 26-Dec-2017 11:13:15
8     transactionType: 'buy'
9         quantity: 75
10         fee: 0
11         price: 71.9000
12
13
14 trans02 =
15
16     Transaction with properties:
17
18         date: 01-Dec-2017 14:30:00
19     transactionType: 'buy'
20         quantity: 100
21         fee: []

```

```

22         price: 7.1900
23
24
25 trans03 =
26
27     Transaction with properties:
28
29         date: 26-Dec-2017 11:13:15
30     transactionType: 'sell'
31         quantity: 25
32         fee: []
33         price: 52
34
35
36 trans04 =
37
38     Transaction with properties:
39
40         date: 26-Dec-2017 11:13:15
41     transactionType: 'sell'
42         quantity: 25
43         fee: 7
44         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 **addTransaction()** method works. The updated class definition looks like this:

```

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 = 'X-ray Yankee Zulu'

```

```

10     symbol = 'XYZ'
11     units = 'shares'
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     function obj = addTransaction(obj, newTransaction)
29         % addTransaction() adds a transaction newTransaction to
           the
30         % transactionList property of obj
31
32         % add the transaction on the end of the list
33         obj.transactionList = [obj.transactionList
           newTransaction];
34
35         % if the length of the list is greater than 1, the list
           may
36         % require sorting
37
38         if length(obj.transactionList) > 1
39             % Create a list of transaction dates by iterating
           through
40             % all transactions and adding dates to dateList
41             dateList = []; % empty date list
42             for TransIdx = 1:n_trans_old+1
43                 % append the date of obj.transactionList(
           TransIdx) to
44                 % dateList (unsorted)
45                 dateList = [dateList ...
           obj.transactionList(TransIdx).date];
46             end
47
48             % sort the transaction list

```

```

51         % obtain an index of sorted transaction dates
52         % The sort function returns the sorted list along
53         % with
54         % the indices of sorted lists within the unsorted
55         % list
56         % The indices of the dateList, sortIndex, will be
57         % used to
58         % sort the list of transactions.
59         [~, sortIndex] = sort(dateList);
60
61         % reorder the unsorted transactions and store in
62         % obj.transactionList
63         obj.transactionList = obj.transactionList(sortIndex)
64         ;
65     end % END: if length(obj.transactionList) > 1
66
67 end
68
69 function obj = listTransactions(obj)
70     numTrans = length(obj.transactionList);
71
72     if numTrans > 0
73         for transIndex = 1:numTrans
74             obj.transactionList(transIndex).listDetails;
75         end
76     else
77         disp(['Asset ', obj.name, ' (', obj.symbol, ...
78             ') has no transactions.'])
79     end
80
81 end
82
83 end % END: methods
84
85 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:

```

1 classdef Transaction
2     %Transaction defines a Transaction class to represent a
      transaction of
3     %an investment asset, represented by the Asset class.
4     %
5
6     properties
7         date % a MATLAB datetime object
8         transactionType % 'buy', 'sell', 'short', 'dividend', 'split
          '
9         quantity
10        fee
11        price
12
13    end % END: properties
14
15    methods
16        function obj = Transaction( varargin )
17            %Transaction constructs an instance of the Transaction
              class.
18            %
19            % SYNTAX:
20            %
21            % newTrans = Transaction( Ttype, Tquantity, Tprice)
22            %     Defines a new transaction with the current date/
              time in
23            %     the date field.
24            %
25            % newTrans = Transaction( Tdate, Ttype, Tquantity,
              Tprice)
26            %     Defines a new transaction with a specified date/
              time. The
27            %     date/time object may be specified as MATLAB
              datetime

```

```

28 %      object or as a 3- or 6-element date vector.
29 %
30 % newTrans = Transaction( Tdate, Ttype, Tquantity,
31 %      Tprice, fee)
32 %      Defines a new transaction with a specified date/
33 %      time and
34 %      a transaction fee.
35 %
36 % This switch-case control group defines the Transaction
37 % object
38 % differently based on the
39 switch nargin
40     % The 3-input case assumes that the date is now
41     case 3 % newTr = Transaction(Ttype, Tquantity,
42         Tprice)
43         obj.date = datetime('now');
44         obj.transactionType = varargin{1};
45         obj.quantity = varargin{2};
46         obj.fee = 0;
47         obj.price = varargin{3};
48
49     % The 4-input case allows the transaction date to be
50     % specified
51     case 4 % newTr = Transaction(Tdate, Ttype, Tquantity
52         , ...
53         Tprice)
54
55         switch class(varargin{1})
56             case 'datetime'
57                 obj.date = varargin{1};
58             case 'double' % the date specifier is in
59                 vector form
60                 % convert a date vector to a datetime
61                 object
62                 obj.date = datetime(varargin{1});
63             otherwise
64                 error('Invalid transaction date
65                     specification.')
66         end
67
68     obj.transactionType = varargin{2};
69     obj.quantity = varargin{3};
70     obj.price = varargin{4};
71
72 case 5
73     % newTr = Transaction( Tdate, Ttype, Tquantity,
74     ...

```

```

67         %                Tprice, Tfee )
68
69         switch class(varargin{1})
70             case 'datetime'
71                 obj.date = varargin{1};
72             case 'double' % the date specifier is in
73                 % convert a date vector to a datetime
74                 % object
75                 obj.date = datetime(varargin{1});
76             otherwise
77                 error('Invalid transaction date
78                     specification.')
79             end
80
81             obj.transactionType = varargin{2};
82             obj.quantity = varargin{3};
83             obj.price = varargin{4};
84             obj.fee = varargin{5};
85
86         otherwise
87             error('Invalid number of input arguments.')
88         end
89     end
90
91     function listDetails(obj)
92         DateString = [char(obj.date) ...
93             blanks(20 - length(char(obj.date))) ];
94         TypeString = [blanks(10-length(obj.transactionType)),
95             ...
96             obj.transactionType];
97
98         QtyString = [blanks(10 - length(num2str(obj.quantity))),
99             ...
100             num2str(obj.quantity)];
101
102         RawPriceStr = num2str(obj.price, '%0.3g');
103         PriceString = [' at ', blanks(10 - length(RawPriceStr)),
104             ...
105             RawPriceStr];
106
107         DetailString = [DateString, TypeString, QtyString,
108             PriceString];
109
110         disp(DetailString)
111     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:

```

1 % testbedAsset.m
2
3 % create an new Asset object with no transactions
4 newAsset = Asset('X-ray Yankee Zulu, Inc.', 'XYZ', 'shares')
5
6 % add a buy transaction with the current date
7 newAsset = newAsset.addTransaction( Transaction('buy', 100, 24.03) )
8
9 % list transaction data after the first addition
10 newAsset.listTransactions;
11
12 % add a transaction with an earlier date
13 newAsset = newAsset.addTransaction( Transaction([2017, 12, 1], ...
14     'buy', 25, 22.97) )
15
16 % list transaction data after the first addition
17 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:

```

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: []
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.

```

1 classdef Transaction
2     %Transaction defines a Transaction class to represent a
      transaction of
3     %an investment asset, represented by the Asset class.
4     %
5
6     properties
7         date % a MATLAB datetime object
8         transactionType % 'buy', 'sell', 'short', 'dividend', 'split
          '
9         quantity = 0;
10        dividend = 0;
11        split_ratio = 1;
12        fee = 0; % Default value: zero
13        price
14
15    end % END: properties
16
17    methods
18        function obj = Transaction( varargin )
19            %Transaction constructs an instance of the Transaction
              class.
20            %
21            % SYNTAX:
22            %
23            % newTrans = Transaction( Ttype, Tquantity, Tprice)
24            %     Defines a new transaction with the current date/
              time in
25            %     the date field. Valid transaction types are 'buy',
26            %     'sell', and 'div-rnv' (dividend reinvestment).
27            %

```

```

28     % newTrans = Transaction( Tdate, Ttype, Tquantity,
    Tprice)
29     %     Defines a new transaction with a specified date/
    time. The
30     %     date/time object may be specified as MATLAB
    datetime
31     %     object or as a 3- or 6-element date vector.
32     %
33     % newTrans = Transaction( Tdate, Ttype, Tquantity,
    Tprice, fee)
34     %     Defines a new transaction with a specified date/
    time and
35     %     a transaction fee.
36     %
37
38     % This switch-case control group defines the Transaction
    object
39     % differently based on the
40     switch nargin
41         % The 3-input case assumes that the date is now
42         case 3 % newTr = Transaction(Ttype, Tquantity,
            Tprice)
43             obj.date = datetime('now');
44             obj.transactionType = varargin{1};
45             obj.quantity = varargin{2};
46             obj.fee = 0;
47             obj.price = varargin{3};
48
49         % The 4-input case allows the transaction date to be
50         % specified
51         case 4 % newTr = Transaction(Tdate, Ttype, Tquantity
            , ...
52             %
53             Tprice)
54
55             switch class(varargin{1})
56                 case 'datetime'
57                     obj.date = varargin{1};
58                 case 'double' % the date specifier is in
                    vector form
59                     % convert a date vector to a datetime
                    object
60                     obj.date = datetime(varargin{1});
61                 otherwise
62                     error('Invalid transaction date
                    specification.')
63
64             end
65
66         obj.transactionType = varargin{2};

```

```

65
66         obj.price = varargin{4};
67
68         switch obj.transactionType
69             case 'div-rnv'
70                 obj.dividend = varargin{3};
71                 obj.quantity = obj.dividend/obj.price;
72             otherwise
73                 obj.quantity = varargin{3};
74         end
75
76
77     case 5
78         % newTr = Transaction( Tdate, Ttype, Tquantity,
79             % ...
80             % Tprice, Tfee )
81
82         switch class(varargin{1})
83             case 'datetime'
84                 obj.date = varargin{1};
85             case 'double' % the date specifier is in
86                 % convert a date vector to a datetime
87                 % object
88                 obj.date = datetime(varargin{1});
89             otherwise
90                 error('Invalid transaction date
91                     specification.')
92         end
93
94         obj.transactionType = varargin{2};
95         obj.quantity = varargin{3};
96         obj.price = varargin{4};
97         obj.fee = varargin{5};
98
99     otherwise
100         error('Invalid number of input arguments.')
101     end
102
103 end
104
105 function listDetails(obj)
106     DateString = [char(obj.date) ...
107         blanks(20 - length(char(obj.date))) ];
108     TypeString = [blanks(10-length(obj.transactionType)),
109         ...
110         obj.transactionType];

```

```

108         QtyString = [blanks(10 - length(num2str(obj.quantity))),
109             ...
110             num2str(obj.quantity)];
111
112         RawPriceStr = num2str(obj.price, '%0.3g');
113         PriceString = [' at ', blanks(10 - length(RawPriceStr)),
114             ...
115             RawPriceStr];
116
117         DetailString = [DateString, TypeString, QtyString,
118             PriceString];
119
120         disp(DetailString)
121     end
122 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:

```

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

```

```

24         obj.symbol = ASymbol;
25         obj.units = AUnits;
26     end
27
28     function obj = addTransaction(obj, newTransaction)
29         % addTransaction() adds a transaction newTransaction to
30         % the
31         % transactionList property of obj
32
33         % add the transaction on the end of the list
34         obj.transactionList = [obj.transactionList
35                                newTransaction];
36
37         % if the length of the list is greater than 1, the list
38         % may
39         % require sorting
40
41         if length(obj.transactionList) > 1
42             % Create a list of transaction dates by iterating
43             % through
44             % all transactions and adding dates to dateList
45             dateList = []; % empty date list
46             for TransIdx = 1:length(obj.transactionList)
47                 % append the date of obj.transactionList(
48                 % TransIdx) to
49                 % dateList (unsorted)
50                 dateList = [dateList ...
51                             obj.transactionList(TransIdx).date];
52             end
53
54             % sort the transaction list
55
56             % obtain an index of sorted transaction dates
57             % The sort function returns the sorted list along
58             % with
59             % the indices of sorted lists within the unsorted
60             % list
61             % The indices of the dateList, sortIndex, will be
62             % used to
63             % sort the list of transactions.
64             [~, sortIndex] = sort(dateList);
65
66             % reorder the unsorted transactions and store in
67             % obj.transactionList
68             obj.transactionList = obj.transactionList(sortIndex)
69             ;
70         end % END: if length(obj.transactionList) > 1
71
72

```

```

63     end
64
65     function obj = listTransactions(obj)
66         numTrans = length(obj.transactionList);
67
68         if numTrans > 0
69             disp(['Transactions for ', obj.name, ' (' , obj.
70                 symbol, ...
71                 '):'])
72             for transIndex = 1:numTrans
73                 obj.transactionList(transIndex).listDetails;
74             end
75         else
76             disp(['Asset ', obj.name, ' (' , obj.symbol, ...
77                 ') has no transactions.'])
78         end
79     end
80 end
81
82 function varargout = calculateValue(obj)
83     % Asset/calculateValue performs an analysis on the list
84     % of
85     % transactions to calculate asset holdings and their
86     % value at
87     % the time of the last transaction.
88     %
89     % Syntax:
90     %
91     % Value = myAsset.calculateValue returns the value of
92     % holdings
93     % at the time of the last transaction.
94     %
95     % [Value, Units] = myAsset.calculateValue additionally
96     % returns
97     % the total number of units held.
98     %
99     % [Value, Units, CostBasis] = myAsset.calculateValue
100    % returns
101    % the investor's cost basis.
102    %
103    Value = 0; % value of holdings
104    TotalUnits = 0; % number of units held
105    CostBasis = 0; % cost basis of investment
106    if ~isempty(obj.transactionList)
107        numTrans = length(obj.transactionList);

```

```

105     % Units: storage vector for units owned as a fcn. of
106         time
107     Units = zeros(1, numTrans);
108     Cost = zeros(1, numTrans);
109     Value = zeros(1, numTrans);
110
111     % Iterate through all transitions
112     for transIdx = 1:numTrans
113         % extract a single transition
114         tempTrans = obj.transactionList(transIdx);
115
116         if transIdx == 1
117             Units(1) = tempTrans.quantity;
118             Cost(transIdx) = tempTrans.price ...
119                 * tempTrans.quantity ...
120                 + tempTrans.fee;
121
122         else
123
124             switch tempTrans.transactionType
125                 case 'buy'
126                     Units(transIdx) = Units(transIdx-1)
127                         ...
128                         + tempTrans.quantity;
129
130                     Cost(transIdx) = tempTrans.price ...
131                         * tempTrans.quantity ...
132                         + tempTrans.fee;
133
134                 case 'sell'
135                     Units(transIdx) = Units(transIdx-1)
136                         ...
137                         - tempTrans.quantity;
138                     Cost(transIdx) = -tempTrans.price
139                         ...
140                         * tempTrans.quantity ...
141                         + tempTrans.fee;
142
143                 case 'split'
144                     Units(transIdx) = Units(transIdx-1)
145                         ...
146                         * tempTrans.split_ratio;
147
148                 case 'div-rnv'
149                     Units(transIdx) = Units(transIdx-1)
150                         ...
151                         + tempTrans.dividend/tempTrans.

```



```

147                                     price;
148
149                                 end
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.

```

1 % testbedAssetv02.m
2
3 % create an new Asset object with no transactions
4 newAsset = Asset('X-ray Yankee Zulu, Inc.', 'XYZ', 'shares');
5
6 % add a buy transaction with the current date
7 newAsset = newAsset.addTransaction( Transaction('sell', 40, 24.03) )
8 ;
9
10 % list transaction data after the first addition
11 newAsset.listTransactions;
12
13 % add a transaction with an earlier date
14 newAsset = newAsset.addTransaction( Transaction([2017, 1, 1], ...
15     'buy', 25, 22.97) );
16
17 % add a transaction with an earlier date
18 newAsset = newAsset.addTransaction( Transaction([2017, 3, 18], ...
19     'div-rnv', 40, 23.58) );
20
21 % add a transaction with an earlier date
22 newAsset = newAsset.addTransaction( Transaction([2017, 5, 24], ...
23     'buy', 100, 25.17) );
24
25 % list transaction data after the first addition
26 newAsset.listTransactions;
27
28 [Value, Holdings, CostBasis] = newAsset.calculateValue;
29 Gain = Value - CostBasis;
30
31 disp([char(10), 'Asset           : ', newAsset.name, ' (', newAsset.
32     symbol, ...
33     ')', char(10), 'Quantity       : ', num2str(Holdings), ...
34     char(10), 'Value           : $', num2str(Value), ...
35     char(10), 'Cost Basis      : $', num2str(CostBasis), ...
36     char(10), 'Unrealized Gains: $', num2str(Gain), ...
37     ' 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.

```

1 >> testbedAssetv02
```

```

2 Transactions for X-ray Yankee Zulu, Inc. (XYZ):
3 29-Dec-2017 20:46:36      sell      40 at      24
4 Transactions for X-ray Yankee Zulu, Inc. (XYZ):
5 01-Jan-2017              buy      25 at      23
6 18-Mar-2017              div-rnv    1.6964 at      23.6
7 24-May-2017              buy     100 at      25.2
8 29-Dec-2017 20:46:36      sell      40 at      24
9
10 Asset           : X-ray Yankee Zulu, Inc. (XYZ)
11 Quantity        : 86.6964
12 Value           : $2083.3134
13 Cost Basis      : $2130.05
14 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.

```

1 classdef Portfolio
2     %PORTFOLIO defines a container class Portfolio for objects of
3     type
4     % Asset. A Portfolio object calculates its own value by
5     calling each
6     % contained Asset object to evaluate and return its individual
7     value.
8     %
9
10    properties
11        name
12        assetList
13    end
14
15    methods
16        function obj = Portfolio(varargin)
17            %Portfolio constructs an Portfolio object.
18            %
19            % SYNTAX:
20            %
21            % myPortfolio = Portfolio creates an empty portfolio.
22            %
23            % myPortfolio = Portfolio( portfolioName, assetArray )
24            % Detailed explanation goes here

```

```

22
23     switch nargin
24         case 0
25             obj.name = 'Default Portfolio';
26             obj.assetList = [];
27
28         case 2
29             obj.name = varargin{1};
30             if strcmp(class(varargin{2}), 'Asset') % input
31                 checking
32                 obj.assetList = varargin{2};
33             else
34                 error('Non-Asset object specified for asset
35                     list.')
36             end
37     end
38 end % END: Portfolio constructor
39
40 function obj = addAssets(obj, additionalAssets)
41     % Portfolio/addAsset adds new Asset objects to the
42     % assetList
43     % property of a Portfolio object.
44     %
45     % SYNTAX:
46     %
47     % myPortfolio = myPortfolio.addAsset( additionalAssets )
48     %
49     if strcmp(class(additionalAssets), 'Asset') % input
50         checking
51         obj.assetList = [obj.assetList additionalAssets];
52     else
53         error('Non-Asset object specified for asset list.')
54     end
55 end % END: Portfolio/addAssets()
56
57 function varargout = calculateValue(obj)
58
59     PortfolioValue = 0;
60     PortfolioData = [];
61
62     if ~isempty(obj.assetList)
63         numAssets = length(obj.assetList);
64         Symbol = cell(numAssets, 1);
65         Value = zeros(numAssets, 1); % storage vector
66         Holdings = Value; % storage vector
67         CostBasis = Value; % storage vector
68         UnitPrice = Value;

```

```

66         for AssetIdx = 1:numAssets
67             tempAsset = obj.assetList(AssetIdx);
68             [Value(AssetIdx), Holdings(AssetIdx), ...
69              CostBasis(AssetIdx), ...
70              UnitPrice(AssetIdx)] = tempAsset.
              calculateValue;
71             Symbol{AssetIdx} = tempAsset.symbol;
72         end
73         UnrealizedGains = Value - CostBasis;
74
75         PortfolioValue = sum(Value);
76         PortfolioData = table(Symbol, Value, Holdings, ...
77                               UnitPrice, CostBasis, UnrealizedGains);
78     end
79
80     switch nargout
81     case 1
82         varargout{1} = PortfolioValue;
83     case 2
84         varargout{1} = PortfolioValue;
85         varargout{2} = PortfolioData;
86     otherwise
87         error('Invalid number of outputs specified.')
88     end
89
90     end % END: Portfolio/calculateValue()
91 end
92 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.

```

1 % testbedPortfolio.m
2
3 % Define firstAsset and add some transactions
4 firstAsset = Asset('X-ray Yankee Zulu', 'XYZ', 'shares');
5 firstAsset = firstAsset.addTransaction( Transaction([2015, 10, 1],

```

```

6     ...
7     'buy', 100, 26.75) );
8 firstAsset = firstAsset.addTransaction( Transaction([2016, 5, 1],
9     ...
10    'sell', 25, 32.18) );
11 firstAsset = firstAsset.addTransaction( Transaction([2016, 12, 28],
12     ...
13    'div-rnv', 75.29, 33.42) );
14 firstAsset = firstAsset.addTransaction( Transaction([2017, 4, 1],
15     ...
16    'buy', 30, 32.18) );
17 firstAsset = firstAsset.addTransaction( Transaction([2017, 12, 27],
18     ...
19    'div-rnv', 82.15, 36.25) );
20
21 secondAsset = Asset('Quebec Romeo Sierra', 'QRS', 'shares');
22 secondAsset = secondAsset.addTransaction( Transaction([2014, 3, 8],
23     ...
24    'buy', 50, 13.28) );
25 secondAsset = secondAsset.addTransaction( Transaction([2014, 12,
26    29], ...
27    'div-rnv', 42.69, 16.24) );
28 secondAsset = secondAsset.addTransaction( Transaction([2015, 6, 24],
29     ...
30    'buy', 20, 17.01) );
31 secondAsset = secondAsset.addTransaction( Transaction([2015, 12,
32    28], ...
33    'div-rnv', 53.12, 17.79) );
34 secondAsset = secondAsset.addTransaction( Transaction([2016, 8, 13],
35     ...
36    'buy', 50, 18.24) );
37 secondAsset = secondAsset.addTransaction( Transaction([2016, 12,
38    27], ...
39    'div-rnv', 62.24, 19.13) );
40
41 newPortfolio = Portfolio; % create an empty Portfolio object
42
43 % add the newly-created assets as
44 newPortfolio = newPortfolio.addAssets([firstAsset, secondAsset]);
45
46 [PortfolioValue, PortfolioData] = newPortfolio.calculateValue;
47 TotalValue = sum(PortfolioData.Value);
48 TotalCostBasis = sum(PortfolioData.CostBasis);
49 TotalGains = sum(PortfolioData.UnrealizedGains);
50 disp(['Total portfolio value: $', num2str(TotalValue), char(10), ...
51     'Cost basis           : $', num2str(TotalCostBasis), char(10),
52     ...

```

```
42 |         '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:

```
1 >> testbedPortfolio
2 Total portfolio value: $6435.3136
3 Cost basis           : $4752.1
4 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.

```
1 classdef Avatar
2     %AVATAR objects represent a player in a multi-player role-
      playing game.
3     % The AVATAR class defines properties typical of avatars in
      RPGs.
4     %
5     % By E.P. Blair
6     % Baylor University
7
8     properties
9         Name = 'Unidentified Player';
10        Class % {'mage', 'warrior', 'thief', 'jedi', 'sith', 'none'}
11        Level = 1; % a numerical rank for a player
12        Position = [0 0]; % (x, y) double for
13        XP = 0; % a numerical property for accumulating experience
          points
14        HealthPointsMax = 100; % player's maximum health points (HP)
15        Vitality = 1 % player's actual vitality (fraction of maximum
          vitality)
16        % HP = Vitality * HealthPointsMax
17        Attack % numerical rating for offensive capabilities
18        Defense % numerical rating for defensive capabilities
19        WeaponList % list of player's a offensive equipment
20        ArmorList % list of player's a defensive equipment
21        EquipmentList % list of player's equipment items
22    end % END: properties
23
24    methods
25        function obj = Avatar(varargin)
26            %Avatar instantiates an AVATAR object
27            % SYNTAX
28            %
29            % newAvatar = Avatar creates a default AVATAR object
30            %
```



```

31         % newAvatar = Avatar(Uname) creates a default
           AVATAR
32         %             object and specifies Uname as the name.
33         %
34         % newAvatar = Avatar(Uname, PlayerClass) creates a
           default
35         %             AVATAR object with name Uname and class
36         %             PlayerClass.
37         %
38
39         switch nargin
40             case 0
41                 obj.Class = 'none';
42             case 1
43                 obj.Name = varargin{1};
44                 obj.Class = 'none';
45             case 2
46                 obj.Name = varargin{1};
47                 obj.Class = varargin{2};
48         end
49     end
50
51     function obj = move(obj, dispVect)
52         % myPlayer = myPlayer.move( [dX dY] ) displaces the
           avatar on
53         % the battlefield.
54         obj.Position = obj.Position + dispVect;
55     end
56
57     function disp(obj)
58         % disp(someAvatar) is the display function for the
           AVATAR
59         % class.
60         disp([obj.Name, ' (Level ', num2str(obj.Level), ' ', ...
61             obj.Class, ') is at (', ...
62             num2str(obj.Position(1)) , ', ', ...
63             num2str(obj.Position(2)), ')'].');
64     end
65
66     end % END: methods
67 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 built-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:

```
1 >> newAvatar = Avatar('Sargon', 'warrior')
2
3 newAvatar =
4
5 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:

```
1 >> newAvatar = newAvatar.move([27, -126])
2
3 newAvatar =
4
5 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.

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

```

3      % The AVATAR class defines properties typical of avatars in
      % RPGs.
4      %
5      % By E.P. Blair
6      % Baylor University
7
8      properties
9
10         <-- snip -->
11
12         % Graphics-related properties
13         Drawing % A struct of handles to the drawing components
14
15     end % END: properties
16
17     methods
18
19         <-- snip -->
20
21         function obj = draw(obj, varargin)
22
23             % Default values
24             TargetAxes = [];
25             % Override default values: parse varargin for property-
                % value
26             % pairs
27
28             args = varargin;
29             while length(args) >= 2
30                 prop = args{1};
31                 val = args{2};
32                 args = args(3:end);
33
34                 switch prop
35                     case 'Axes'
36                         TargetAxes = val;
37                     otherwise
38                         error(['''', prop, '' is an invalid ', ...
39                             'property specifier.'])
40                 end % END switch prop
41             end % END while length(args) >= 2
42
43             if isempty(obj.Drawing)
44
45                 % DRAW THE SQUARE (MAIN BODY)
46                 % Calculate relative points of corners
47                 x_rel = [-5 -5 5 5];
48                 y_rel = [-5 5 5 -5];

```

```

49 % Calculate absolute points of corners
50 x = obj.Position(1) + x_rel;
51 y = obj.Position(2) + y_rel;
52 % visualization
53 Drawing.Body = patch(x, y, [1 1 1], ...
54     'EdgeColor', [0 0 0], 'LineWidth', 2);
55
56 NameText = text(obj.Position(1), obj.Position(2),
57     ...
58     obj.Name, 'FontName', 'Times', 'FontSize', 18,
59     ...
60     'HorizontalAlignment', 'center', ...
61     'VerticalAlignment', 'bottom');
62
63 % PLAYER INFO STRING
64 % 'LVL. X C' (X = Level, C = Class)
65 PlayerInfoStr = ['LVL. ', num2str(obj.Level), ' ',
66     ...
67     upper(obj.Class(1:3))];
68 xPInfoStr = obj.Position(1);
69 yPInfoStr = obj.Position(2) + 4;
70 PlayerInfoText = text(xPInfoStr, yPInfoStr,
71     PlayerInfoStr, ...
72     'FontName', 'Times', 'FontSize', 14, ...
73     'HorizontalAlignment', 'center', ...
74     'VerticalAlignment', 'middle');
75
76 % HEALTH BAR
77 % Full health will span [-4, 4] (relative)
78 % No health is a point at -4 (relative
79 % Color will transition from [0 0.5 0] (green, full)
80 % to [1 0 0] (red, empty)
81 cHealthLine = [ 1-obj.Vitality, 0.75*obj.Vitality,
82     0];
83 xHealthLine = obj.Position(1) + [-4, -4+8*obj.
84     Vitality];
85 yHealthLine = obj.Position(2) - [2 2];
86 HealthLine = line( xHealthLine, yHealthLine, ...
87     'Color', cHealthLine, ...
88     'LineWidth', 5);
89
90 % HEALTH STATUS STRING (HealthStr)
91 % 'HP: XX/MAX'
92 HealthStr = ['HP: ', ...
93     num2str(round(obj.Vitality*obj.HealthPointsMax))
94     , ...
95     '/', num2str(obj.HealthPointsMax)];

```

```

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

```

1 % testbedAvatar.m
2 close all;
3 clear all;
4
5 Player1 = Avatar('Sargon', 'warrior') % create Player1
6
7 Player1 = Player1.move([-3, 13]) % displace Player1
8
9 Player1 = Player1.draw; % execute draw()
10
11 xlim([-20, 20]); % adjust the x limits beyond the square patch
12 axis equal % make the x- and y-scales equal

```

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

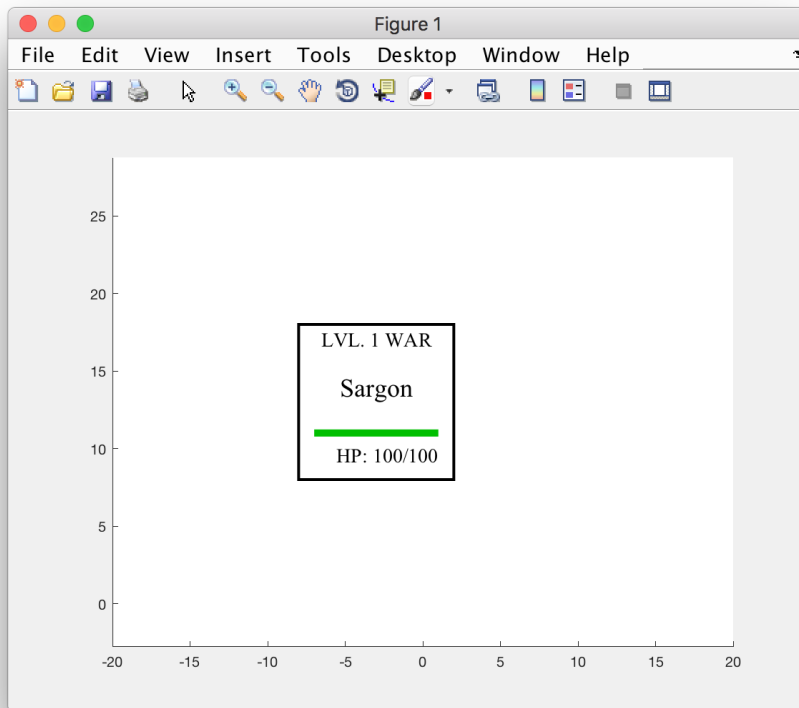


Figure 6: A rudimentary graphical representation for an *Avatar* object. Here, *Player1* has the name 'Sargon' and is drawn to indicate level 1 status as a warrior with full health (100/100 HP).

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:

```
1 if ctrl_expr
2     statements
3 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

```
1 % basic_if_control.m
2
3 a = true
4 if a
5     disp('a is true.')
6 end
```

The output of **basic_if_control.m** is

```
1 >> basic_if_control
2
3 a =
4
5     logical
6
7     1
8
9 '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()**:

```
1 function sign_fun(x)
2 % sign_fun(x) prints a message about the negativity of x.
3 %
4 % By E.P. Blair
5 % Baylor University
6
7 if x < 0
8     disp('x is negative.')
9 else
10     disp('x is non-negative.')
11 end
```


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

```
1 >> sign_fun(-5)
2 x is negative.
3 >> sign_fun(-1.25)
4 x is negative.
5 >> sign_fun(0)
6 x is non-negative.
7 >> sign_fun(1)
8 x is non-negative.
9 >>
```

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()`:

```
1 function sign_fun(x)
2 % sign_fun(x) prints a message about the sign of x.
3 %
4 % By E.P. Blair
5 % Baylor University
6
7 if x < 0
8     disp('x is negative.')
9 elseif x > 0
10    disp('x is positive.')
11 else
12    disp('x is zero.')
13 end
```

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

```
1 >> sign_fun(-2)
2 x is negative.
3 >> sign_fun(3)
4 x is positive.
5 >> sign_fun(0)
6 x is zero.
```

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

```
1 function lg = letter_grade(percentScore)
2 % letter_grade maps a percentage score to a letter grade.
3
4 if percentScore < 60
5     lg = 'F';
6 elseif percentScore < 70
7     lg = 'D';
8 elseif percentScore < 80
9     lg = 'C';
```

```

10 elseif percentScore < 90
11     lg = 'B';
12 else
13     lg = 'A';
14 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:

```

1 switch expr
2     case expr_1
3         statement_group_1
4     case expr_2
5         statement_group_2
6     ...
7     otherwise
8         statement_group_otherwise
9 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.