An Introduction to Object-oriented Programming in MATLAB

Enrique P. Blair, Ph.D.

January 12, 2018

Contents

1 Introduction to Object-oriented Programming in MATLAB

1.1 Object-oriented Programming

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

1.2 Data Types

1.2.1 Elementary Data Types

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

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

In MATLAB, arrays are formed by concatenating elements within brackets (`[` and `]`). Horizontal concatenation is achieved by grouping elements within brackets but delimiting (separating) them either by commas (`,`) or by white space. Vertical concatenation is achieved by delimiting constituents by a semicolon (`;`). Examples of this are given in the command-line sample below:

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

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

In line 1 of Listing 1, we define `A` as the horizontal concatenation of 1, 2, and 3; and `B` as the horizontal concatenation of 4, 5, and 6. Then, `C` is formed by horizontally concatenating `A` and `B`. The output of line 1 is shown in lines 3-5. Finally, in line 7, `D` is formed by vertically concatenating `A` and `B`, with output on lines 9-12. Similar concatenation may be achieved using strings. Similar concatenations may be done using data of type `char`. To form an array, all elements have the same data type and the same size in memory.

MATLAB also has another type, known as a `cell` (short for "cell array"), in which each element may be of a different type or of different sizes in memory. MATLAB `double` arrays of different

sizes, `char` arrays different sizes, and even other `cell`s may be elements of the same `cell`.

1.2.2 Structures

MATLAB allows the formation of **structures** (`struct`). A structure is a composite variable constituted of **members** of elementary data types. In OOP, member data are called **fields** (the term favored in MATLAB documentation and syntax), or **tags**. For example, the following input (line 1) defines a `struct` `S` with two members, `a` and `b`.

```
1 >> S.a = 7; S.b = 'life'
2
3 S =
4
5   struct with fields:
6
7     a: 7
8     b: 'life'
```

Here, the field `a` of `S`, syntactically referred to as `S.a`, stores a double value of 7; and the field `S.b` stores the `char` array `'life'`.

`struct`s are particularly useful for storing various pieces of information associated with one system or even 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 `struct`s 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.

1.3.1 Writing 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.

1.3.2 The `varargin` Key Word

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.

1.3.3 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.

1.3.4 The `length()` Function

The `length()` function returns the number of elements in an array.

1.3.5 The `disp()` Function

The `disp()` function writes a string to the MATLAB Command Window output.

1.3.6 The `num2str()` Function

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

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.

1.4 Classes

1.4.1 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 `struct`s 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 define operations on objects.

1.4.2 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 behavior, 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 objects of `Asset`.

1.4.3 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 behavior function. Also, if we must modify the behavior, 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 behavior.

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

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

2 Defining Classes in MATLAB

2.1 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. 1. 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. 1):

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

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

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

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

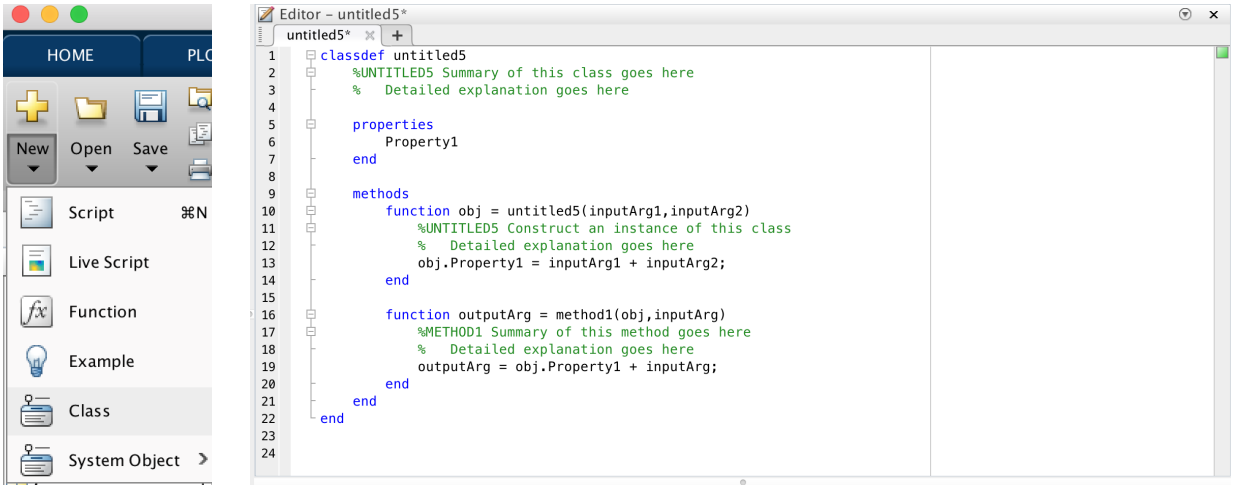
     The body of the `properties` section is used to define class properties (also known as "fields", or "member data"). Here, there is one dummy property defined: `Property1`.

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

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

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

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



(a) (b)

Figure 1: (a) The "New Document" button provides a "Class" option. (b) A new class template obtained in MATLAB 2017b.

Example 2.1. 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 4: MATLAB Command Window input utilizing and testing the `untitled5` class definition of Fig. 1.

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 2.2. Invoke the method `method1()` on the object `myObj` created in Example 2.1.
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
```

2.1.1 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 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`. Then, `method2()` returns a modified copy of `myObj`, which is used to overwrite the original, unmodified object `myObj`.

3 Extended Example: Developing Software to Track Investments

3.1 Developing an `Asset` Class

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

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

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

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

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

```
% testbedAsset.m

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

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

```
>> testbedAsset

newAsset =

  Asset with properties:

                name: 'X-ray Yankee Zulu, Inc.'
              symbol: 'XYZ'
               units: 'shares'
    transactionList: []
```

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

3.2 Developing a `Transaction` Class

Now we will continue to develop the extended example started in Sect. 3.1 by developing a `Transaction` class to have the following properties: `date`, `transactionType`, `quantity`, `price` and `fees`. Here, `date` can store an object of the pre-defined MATLAB class `datetime`; `transactionType` can store a `char` string to identify whether a transaction is a `'buy'`, `'sell'`, `'short'`, etc.; `quantity` and `price` can store a numerical value for the number of units transacted and the unit price, respectively; and, `fees` can be used to store transaction fees.
Solution.

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

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

    end % END: properties

    methods
```

```
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,
                    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
                               vector form
55                              % convert a date vector to a datetime
                                   object
56                              obj.date = datetime(varargin{1});
57                          otherwise
58                              error('Invalid transaction date
                                   specification.')
59                      end
60
61                      obj.transactionType = varargin{2};
62                      obj.quantity = varargin{3};
63                      obj.price = varargin{4};
64
65                  case 5
66                      % newTr = Transaction( Tdate, Ttype, Tquantity,
                           ...
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
                               vector form
73                              % convert a date vector to a datetime
                                   object
74                              obj.date = datetime(varargin{1});
75                          otherwise
76                              error('Invalid transaction date
                                   specification.')
77                      end
78
79                      obj.transactionType = varargin{2};
80                      obj.quantity = varargin{3};
81                      obj.price = varargin{4};
82                      obj.fee = varargin{5};
83
84                  otherwise
85                      error('Invalid number of input arguments.')
86              end
87
88          end
89
90
91
92      end % END: methods
93  end
```

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

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

```
% testbedTransaction.m

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

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

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

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

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

```
>> testbedTransaction

trans01 =

  Transaction with properties:

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


trans02 =

  Transaction with properties:

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

```
               price: 7.1900


trans03 =

  Transaction with properties:

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


trans04 =

  Transaction with properties:

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

3.3 Adding Transactions to an `Asset` Object

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

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

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

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

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

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

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

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

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

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

                % sort the transaction list

```

```
51                  % obtain an index of sorted transaction dates
52                  %   The sort function returns the sorted list along
                        with
53                  %   the indices of sorted lists within the unsorted
                        list
54                  %   The indices of the dateList, sortIndex, will be
                        used to
55                  %   sort the list of transactions.
56                  [~, sortIndex] = sort(dateList);
57
58                  % reorder the unsorted transactions and store in
59                  % obj.transactionList
60                  obj.transactionList = obj.transactionList(sortIndex)
                        ;
61              end % END: if length(obj.transactionList) > 1
62
63          end
64
65          function obj = listTransactions(obj)
66              numTrans = length(obj.transactionList);
67
68              if numTrans > 0
69                  for transIndex = 1:numTrans
70                      obj.transactionList(transIndex).listDetails;
71                  end
72              else
73                  disp(['Asset ', obj.name, ' (', obj.symbol, ...
74                      ') has no transactions.'])
75              end
76
77          end
78
79
80      end % END: methods
81  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 assset, 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,
                    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
                                vector form
55                              % convert a date vector to a datetime
                                    object
56                              obj.date = datetime(varargin{1});
57                          otherwise
58                              error('Invalid transaction date
                                    specification.')
59                      end
60
61                      obj.transactionType = varargin{2};
62                      obj.quantity = varargin{3};
63                      obj.price = varargin{4};
64
65                  case 5
66                      % newTr = Transaction( Tdate, Ttype, Tquantity,
                            ...
```

```
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
                           vector form
73                          % convert a date vector to a datetime
                               object
74                          obj.date = datetime(varargin{1});
75                      otherwise
76                          error('Invalid transaction date
                               specification.')
77                  end
78
79                  obj.transactionType = varargin{2};
80                  obj.quantity = varargin{3};
81                  obj.price = varargin{4};
82                  obj.fee = varargin{5};
83
84              otherwise
85                  error('Invalid number of input arguments.')
86          end
87
88      end
89
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              obj.transactionType];
96
97          QtyString = [blanks(10 - length(num2str(obj.quantity))),
                ...
98              num2str(obj.quantity)];
99
100         RawPriceStr = num2str(obj.price, '%0.3g');
101         PriceString = [' at ', blanks(10 - length(RawPriceStr)),
                ...
102             RawPriceStr];
103
104         DetailString = [DateString, TypeString, QtyString,
                PriceString];
105
106         disp(DetailString)
107     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 5:

```
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 5: 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: [1 1 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: [1 2 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 6: The output of `testbedAsset.m`

Line 3 of Listing 5 resulted in output lines 3-10. Here, we see that `newAsset` has an empty `transactionList` array property. Line 7 of Listing 5 adds a new transaction, resulting in output lines 13-20 here. This shows that `newAsset` now has one transaction. Line 10 of Listing 5 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 5. 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.

3.4 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.

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

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

    end % END: properties

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

```
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                      %               Tprice)
53
54                      switch class(varargin{1})
55                          case 'datetime'
56                              obj.date = varargin{1};
57                          case 'double' % the date specifier is in
                                vector form
58                              % convert a date vector to a datetime
                                    object
59                              obj.date = datetime(varargin{1});
60                          otherwise
61                              error('Invalid transaction date
                                    specification.')
62                      end
63
64                      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                  %           Tprice, Tfee )
80
81                  switch class(varargin{1})
82                      case 'datetime'
83                          obj.date = varargin{1};
84                      case 'double' % the date specifier is in
                           vector form
85                          % convert a date vector to a datetime
                               object
86                          obj.date = datetime(varargin{1});
87                      otherwise
88                          error('Invalid transaction date
                               specification.')
89                  end
90
91                  obj.transactionType = varargin{2};
92                  obj.quantity = varargin{3};
93                  obj.price = varargin{4};
94                  obj.fee = varargin{5};
95
96              otherwise
97                  error('Invalid number of input arguments.')
98          end
99
100     end
101
102
103     function listDetails(obj)
104         DateString = [char(obj.date) ...
105             blanks(20 - length(char(obj.date)) ) ];
106         TypeString = [blanks(10-length(obj.transactionType)),
                ...
107             obj.transactionType];
```

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

Listing 7: 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
          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:length(obj.transactionList)
43                      % append the date of obj.transactionList(
                            TransIdx) to
44                      % dateList (unsorted)
45                      dateList = [dateList ...
46                          obj.transactionList(TransIdx).date];
47                  end
48
49                  % sort the transaction list
50
51                  % obtain an index of sorted transaction dates
52                  %   The sort function returns the sorted list along
                        with
53                  %   the indices of sorted lists within the unsorted
                        list
54                  %   The indices of the dateList, sortIndex, will be
                        used to
55                  %   sort the list of transactions.
56                  [~, sortIndex] = sort(dateList);
57
58                  % reorder the unsorted transactions and store in
59                  % obj.transactionList
60                  obj.transactionList = obj.transactionList(sortIndex)
                        ;
61              end % END: if length(obj.transactionList) > 1
62
```

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

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

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

Listing 8: The `Asset` class with a new `calculateValue()` method.

The new `Asset/calculateValue()` method is listed in lines 82-174 of Listing 8. 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 9). This defines an `Asset` object `newAsset`. It adds several transactions to `newAsset` and lists them using the `Asset` method `listTransactions()`. Finally, the script invokes the `Asset/calculateValue()` method and lists data calculated for this asset. Here, one `disp()` command was used, and the command `char(10)` embeds character ten (the MATLAB code for a newline character) in the string and breaks the string up for readability.

```
% testbedAssetv02.m

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

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

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

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

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

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

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

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

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

Listing 9: A testbed function for `Asset` value calculations.

Running the testbed script of Listing 9 yields the output of Listing 10.

```
>> testbedAssetv02
```

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

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

Listing 10: Output for the testbed function of Listing 9.

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

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

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

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

    properties
        name
        assetList
    end

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

```
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
                           checking
31                          obj.assetList = varargin{2};
32                      else
33                          error('Non-Asset object specified for asset
                               list.')
34                      end
35
36              end
37          end % END: Portfolio constructor
38
39          function obj = addAssets(obj, additionalAssets)
40              % Portfolio/addAsset adds new Asset objects to the
                   assetList
41              %   property of a Portfolio object.
42              %
43              % SYNTAX:
44              %
45              % myPortfolio = myPortfolio.addAsset( additionalAssets )
46              %
47              if strcmp(class(additionalAssets), 'Asset') % input
                   checking
48                  obj.assetList = [obj.assetList additionalAssets];
49              else
50                  error('Non-Asset object specified for asset list.')
51              end
52          end % END: Portfolio/addAssets()
53
54          function varargout = calculateValue(obj)
55
56              PortfolioValue = 0;
57              PortfolioData = [];
58
59              if ~isempty(obj.assetList)
60                  numAssets = length(obj.assetList);
61                  Symbol = cell(numAssets, 1);
62                  Value = zeros(numAssets, 1); % storage vector
63                  Holdings = Value; % storage vector
64                  CostBasis = Value; % storage vector
65                  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 11: The class definition for `Portfolio`*, a container class for objects of class* `Asset`*.*

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

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

Listing 12: The class definition for `Portfolio`, a container class for objects of class `Asset`.

The output for Listing 12 is shown below in Listing 13:

```
1 >> testbedPortfolio
2 Total portfolio value: $6435.3136
3 Cost basis           : $4752.1
4 Unrealized gains     : $1683.2136
```

Listing 13: Output generated by `testbedPortfolio.m`, listed in Listing 12.

4 Graphics in MATLAB

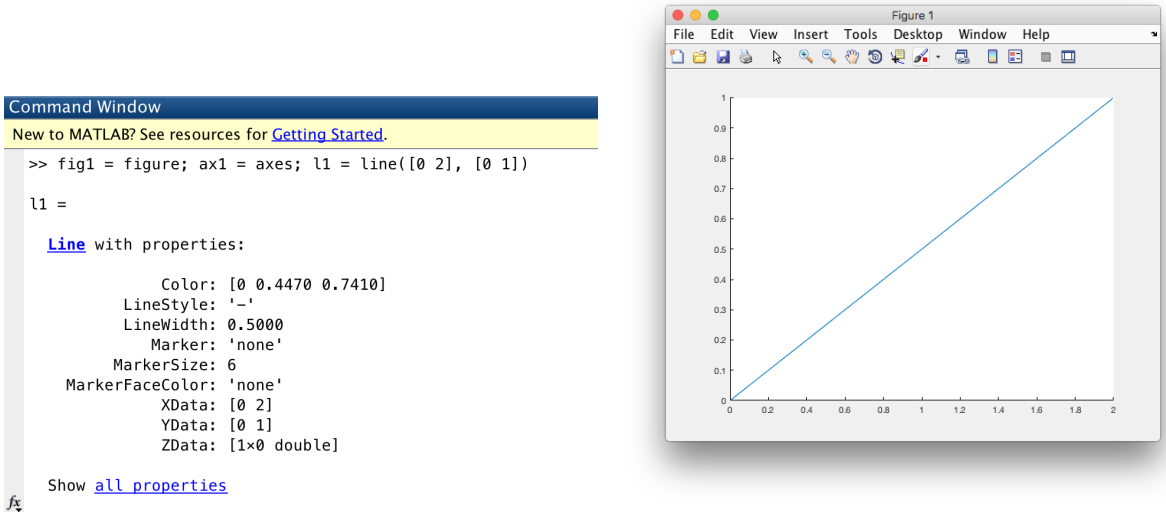
In MATLAB, graphical objects are plotted in `axes` objects, which are contained in `figure` objects. This defines a hierarchy: `figure` objects are at the top, and graphical objects are at the bottom. Objects within this hierarchy have their own properties and functions. For example, each `figure` has a `Children` property which lists all `axes` objects. Thus, the `figure` object is said to be the **parent** object of the `axes` object(s), which are said to be the **children** objects. In fact, child objects such as `axes` objects have a `Parent` property, which identifies each `axes` object's `parent` figure.

A new figure may be instantiated using the `figure` command, and a new axes may be created using the `axes` command. Often, it is helpful assign a **handle variable** to graphing objects so that we can reference them and modify them within a script or program. An example of this is:

```
1 >> fig1 = figure; ax1 = axes; l1 = line([0 2], [0 1])
```

Listing 14: A Command Window input to create a line with handle `l1` within `axes` object `ax1` within `figure` object `fig1`.

The code of Listing 14 produces not only some textual Command Window output (see Fig. 2(a)), but also the figure window shown in Fig. 3. The `line` object `l1` has several properties, and many of



(a) (b)

Figure 2: Command Window output and graphical output of code of Listing 14.

these are suppressed in the Command Window output. Some important properties include `XData`, `YData`, `LineWidth`. Recall that we invoked the `line` constructor using `line([0 2], [0 1])`. The first input array was a set of x-points and was stored in `l1.XData`. The second input array was a set of y-values, stored in `l1.YData`. Thus, the defining points for the `line` `l1` are given by making (x, y) pairs from the values stored in these fields, and the line goes from (`XData(1)`, `YData(1)`) $= (0, 0)$ to (`XData(2)`, `YData(2)`) $= (2, 1)$.

4.1 The Current `figure` and Current `axes` Objects

Since `fig1` is the most recently-created `figure` object, it is referenced in MATLAB as the current figure. We can refer to it by the handle we created, `fig1`, or by the function `gcf` (get current

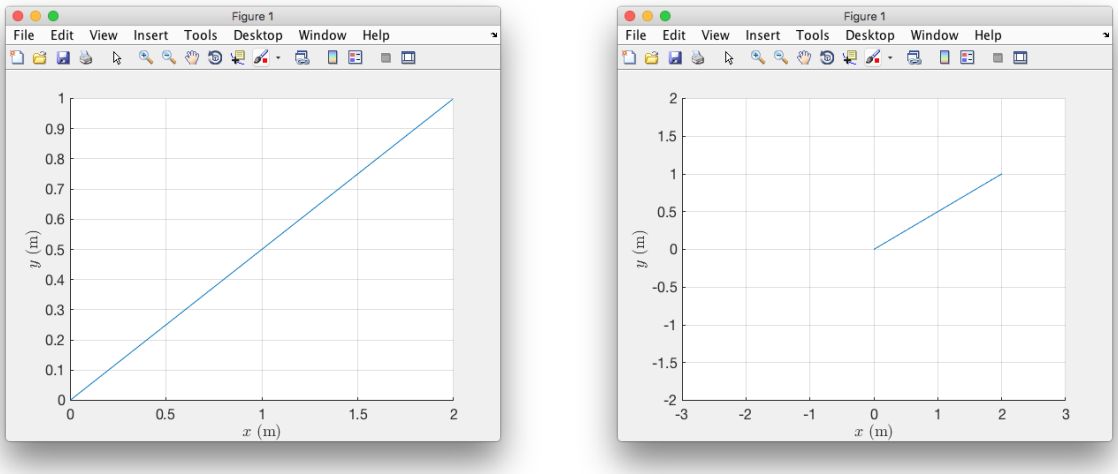
figure), which returns a handle to the current `figure`. Similarly, `ax1` is the current `axes` object, which also may be referenced using `gca` (get current axes) or by `ax1`.

We can use the command `grid on` to toggle on grid lines for the current `axes`. Similarly, we can issue new plot commands, and they will generate new graphics objects as children of the current axes. We also can add x-axis and y-axis labels to the current `axes` by using the `xlabel` and `ylabel`. For example, consider Listing 15:

```
>> grid on; ax1.FontSize=16; xlbl = xlabel('$x$ (m)');  ylbl =
   ylabel('$y$ (m)'); xlbl.Interpreter='latex'; ylbl.Interpreter='
   latex';
```

Listing 15: Command Window input to modify the ***ax1****, the current axes object.*

First, this input instructs MATLAB to display the grid; then, we enlarge the font for `ax1`; next, we define a variable `xlbl` as a handle for the `xlabel` object, which is assigned as a child of `ax1`. Similarly, `ylbl` is a handle for the `ylabel` object (also a child of `ax1`). The strings we provided within the `xlabel()` and `ylabel()` constructor methods enclose a symbol in dollar signs (`$`). This is a fancy technique to make text appear as a mathematical expression as typeset in many textbooks. In order for this to work correctly, the appropriate object must have as it `Interpreter` property set to the value `'latex'`. We accomplish this is the remainder of the command in Listing 15. The graphical result of Listing 15 is shown in Fig. 3(a). Compare Figs. 2(b) and 3(a) to see how we



(a) (b)

Figure 3: Command Window output and graphical output of code of Listing 14 [subfigure (a)] and Listing 15 [subfigure (b)]. In 3(a), we have added a grid and axes labels. In 3(b), we have adjusted the axes limits.

modified `ax1`.

4.1.1 Adjusting the Limits of the Current `axes`

It often is helpful to adjust the x- and y-limits of the current `axes`. To do this, we can use the `xlim()` and `ylim()` functions. When invoked with no arguments, these functions return (get) a 1×2 array describing the x- or y-limits, but when invoked with a single 1×2 vector of the form

`[vlo vhi]`, these functions set the applicable axes limits. As an example, we set the axes limits to be $[-3, 3]$ in the x-direction and $[-2, 2]$ in the y-direction using the following command:

```
>> xlim([-3 3]); ylim([-2 2])
```

Listing 16: A Command Window input to set the axes limits for the current `axes` object.

This creates a larger graphical context for the line segment we drew, as shown in Fig. 3(b).

4.2 Some Basic Graphic Types

4.2.1 The `line` Class

We already have seen the `line` class, which we instantiated using the `line()` constructor method. To modify the line's end points, we can simply modify the `line` object's `XData` and `YData` properties. As an example, we can create a line using a script such as this `basicLine.m` script:

```
% basicLine.m

a = 1; b = 2; c = 3; d = 4;
L0 = line([a b], [c d], 'LineWidth', 2);
```

Listing 17: Listing of the script `basicLine.m`.

Upon running this script, we will have the `line` object `L0` saved in memory, and it can easily be seen that the `XData` property stores the vector `[1, 2]`, and that `YData` stores the vector `[3, 4]`,

```
>> L0.XData

ans =

     1     2

>> L0.YData

ans =

     3     4
```

Listing 18: A Command Window verification of the `XData` and `YData` properties of the `L0` object created in Listing 17.

Also, we can use the `plot()` command to specify `line` objects with two endpoints and many connected data points. To learn the syntax of the `plot()` command, either do an Internet search, or type `doc plot` or `help plot` in the MATLAB Command Window.

We can make several modifications to the `line` object by changing its properties. We list some important properties here:

- `XData`, `YData`, and `ZData` can be modified to change a 2D or 3D line object (yes, MATLAB can make 3D plots).
- `LineWidth` can be used to change the thickness of the line.
- `LineStyle` can be used to set the style of the line.

- `Marker` can be used to specify a marker for data points.

To learn about the many options available to you, type `doc line`.

4.2.2 The `patch` Class

A MATLAB `patch` draws a closed polygon with straight edges. The polygon is specified by specifying the x-, y-, and optional z values of each vertex. The polygon is closed by connecting the last vertex to the first.

The basic syntax for the `patch` constructor 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 x-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
```

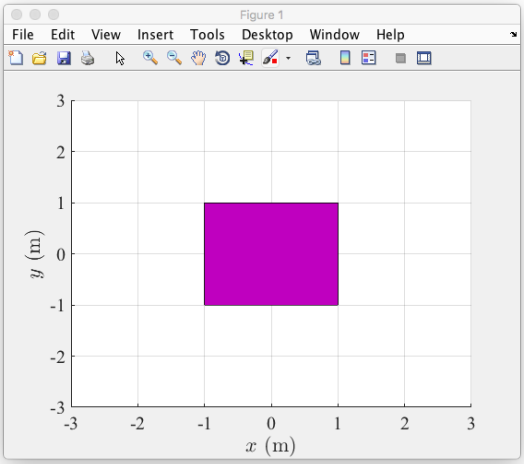
Listing 19: Listing of the script `basicPatch.m`.

The output of the `basicPatch.m` script is shown in Fig. 4.

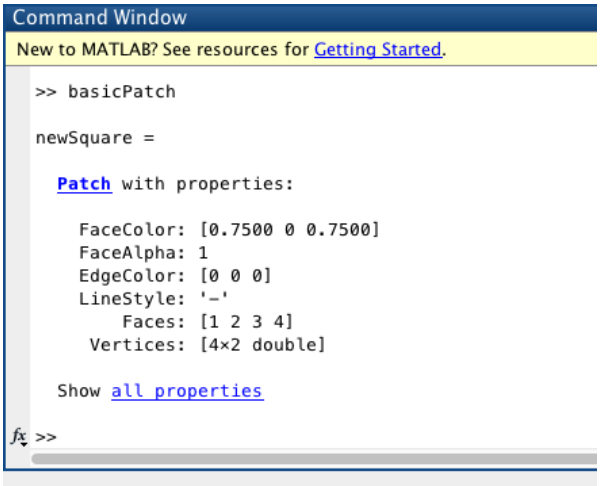
The `patch` object is centered at the origin in Fig. 4(a). An example of a simple modification to this is to translate the patch by the vector $(\Delta x, \Delta y)$. To do this, we define Δx and Δy, and add these to the original `XData` and `YData` properties of the `newSquare` object. This can be done 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 20: A Command Window modification to the `patch` defined in Listing 19. 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.



(a)



(b)

Figure 4: Graphical and Command Window output of the `basicPatch.m` *script of Listing 19.*

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. 5. Similar modifications may be made to a `line`-class object.

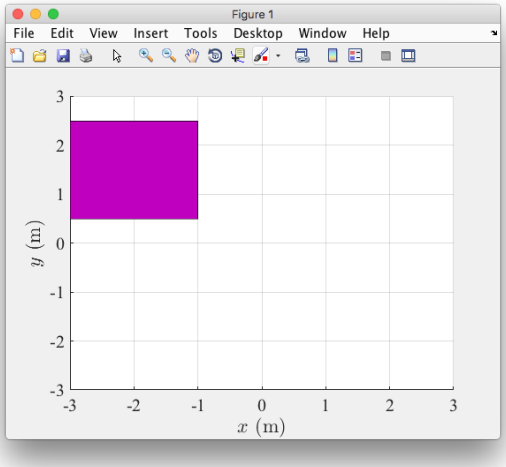


Figure 5: The `patch` *object created using* `basicPatch.m` *is modified by using the* `set()` *function using the code of Listing 20. The resulting* `patch` *no longer sits at the origin.*

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

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 21. 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 densive 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(Username) creates a default
                    AVATAR
32              %           object and specifies Username as the name.
33              %
34              %   newAvatar = Avatar(Username, PlayerClass) creates a
                    default
35              %            AVATAR object with name Username 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 21: An initial class definition file for the ***Avatar*** *class.*

5.1.1 Overriding the `disp()` Function

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

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

As an example, see Listing 22, where we test the new `Avatar` class definition in the Command Window. When the `Avatar` constructor is invoked (and not suppressed), the result is an object of class `Avatar`, so the `Avatar` class `disp()` method is invoked to display information about the resulting `Avatar` object:

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

newAvatar =

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

Listing 22: Command Window input and output to test the `Avatar` class constructor and overridden `disp()` method.

5.1.2 Testing the `move()` Method

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

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

newAvatar =

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

Listing 23: 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 24 shows the new snippets of the `Avatar` class definition file.

```
classdef Avatar
    %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                      obj.Name, 'FontName', 'Times', 'FontSize', 18,
                            ...
58                      'HorizontalAlignment', 'center', ...
59                      'VerticalAlignment', 'bottom');
60
61                  % PLAYER INFO STRING
62                  % 'LVL. X C' (X = Level, C = Class)
63                  PlayerInfoStr = ['LVL. ', num2str(obj.Level), ' ',
                        ...
64                      upper(obj.Class(1:3))];
65                  xPInfoStr = obj.Position(1);
66                  yPInfoStr = obj.Position(2) + 4;
67                  PlayerInfoText = text(xPInfoStr, yPInfoStr,
                        PlayerInfoStr, ...
68                      'FontName', 'Times', 'FontSize', 14, ...
69                      'HorizontalAlignment', 'center', ...
70                      'VerticalAlignment', 'middle');
71
72
73                  % HEALTH BAR
74                  % Full health will span [-4, 4] (relative)
75                  % No health is a point at -4 (relative
76                  % Color will transition from [0 0.5 0] (green, full)
77                  % to [1 0 0] (red, empty)
78                  cHealthLine = [ 1-obj.Vitality, 0.75*obj.Vitality,
                        0];
79                  xHealthLine = obj.Position(1) + [-4, -4+8*obj.
                        Vitality];
80                  yHealthLine = obj.Position(2) - [2 2];
81                  HealthLine = line( xHealthLine, yHealthLine, ...
82                      'Color', cHealthLine, ...
83                      'LineWidth', 5);
84
85                  % HEALTH STATUS STRING (HealthStr)
86                  % 'HP: XX/MAX'
87                  HealthStr = ['HP: ', ...
88                      num2str(round(obj.Vitality*obj.HealthPointsMax))
                            , ...
89                      '/', num2str(obj.HealthPointsMax)];
```

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

Listing 24: 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 25) is the MATLAB `figure` shown in Fig. 6. At this point, the `draw()` method seems to work as well as designed. Next steps include:

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

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

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

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

Player1 = Player1.draw; % execute draw()

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

Listing 25: A testbed function tests the new `draw()` method.

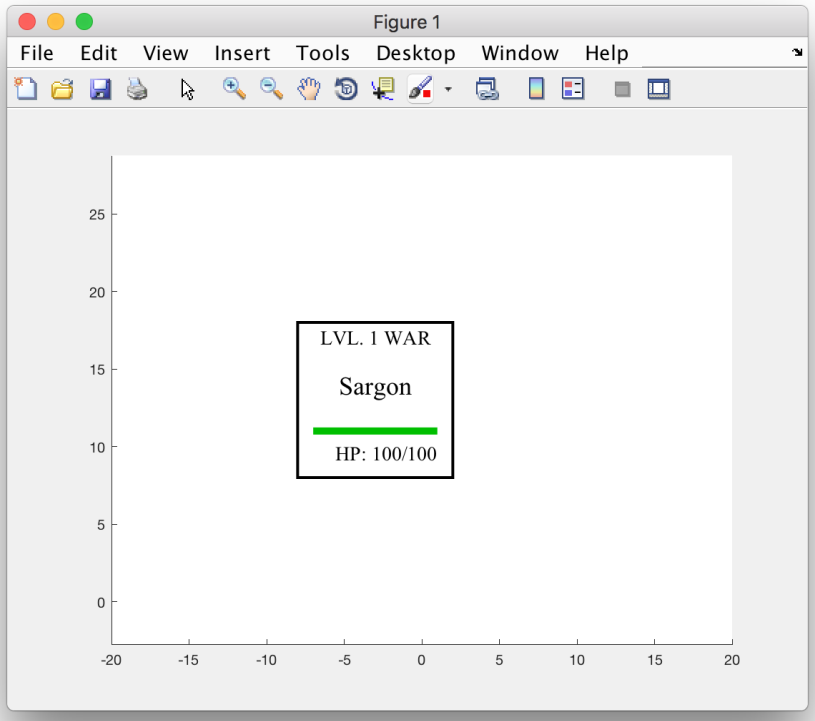


Figure 6: A rudimentary graphical representation for an `Avatar` *object. Here,* `Player1` *has the name* `'Sargon'` *and is drawn to indicate level 1 status as a warrior with full health (100/100 HP).*

A Control Statements

A.1 `if-elseif-else-end`

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

```
if ctrl_expr
  statements
end
```

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

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

```
% basic_if_control.m

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

The output of `basic_if_control.m` is

```
>> basic_if_control

a =

  logical

   1

'a' is true.
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A.2 `for` Loops

A.3 `switch-case` Controls

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

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

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