MATLAB Classes and Objects

This chapter describes how to define your own classes in MATLAB. Classes and objects enable you to add new data types and new operations to MATLAB. The *class* of a variable describes the structure of the variable and indicates the kinds of operations and functions that can apply to the variable. An *object* is an instance of a particular class. The phrase *object-oriented programming* describes an approach to writing programs that emphasizes the use of classes and objects.

Classes and Objects: An Overview

All MATLAB data types are implemented as object-oriented classes. You can add data types of your own to your MATLAB environment by creating additional classes. These user-defined classes define the structure of your new data type, and the M-file functions, or *methods*, that you write for each class define the behavior for that data type.

These methods can also define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the new data types. For example, a class called `polynomial` might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials.

With MATLAB classes you can

- Create methods that override existing MATLAB functionality
- Restrict the operations that are allowed on an object of a class
- Enforce common behavior among related classes by inheriting from the same parent class
- Significantly increase the reuse of your code

Read more about MATLAB classes in Classes and Objects.

You can view classes as new data types having specific behaviors defined for the class. For example, a polynomial class might redefine the addition operator (+) so that it correctly performs the operation of addition on polynomials. Operations defined to work with objects of a particular class are known as *methods* of that class.

You can also view classes as new items that you can treat as single entities. An example is an arrow object that MATLAB can display on graphs (perhaps composed of MATLAB line and patch objects) and that has properties like a Handle Graphics object. You can create an arrow simply by instantiating the arrow class.

You can add classes to your MATLAB environment by specifying a MATLAB structure that provides data storage for the object and creating a class directory containing M-files that operate on the object. These M-files contain the methods for the class. The class directory can also include functions that define the way various MATLAB operators, including arithmetic operations, subscript referencing, and concatenation, apply to the objects. Redefining how a built-in operator works for your class is known as *overloading* the operator.

Features of Object-Oriented Programming

When using well-designed classes, object-oriented programming can significantly increase code reuse and make your programs easier to maintain and extend. Programming with classes and objects differs from ordinary structured programming in these important ways:

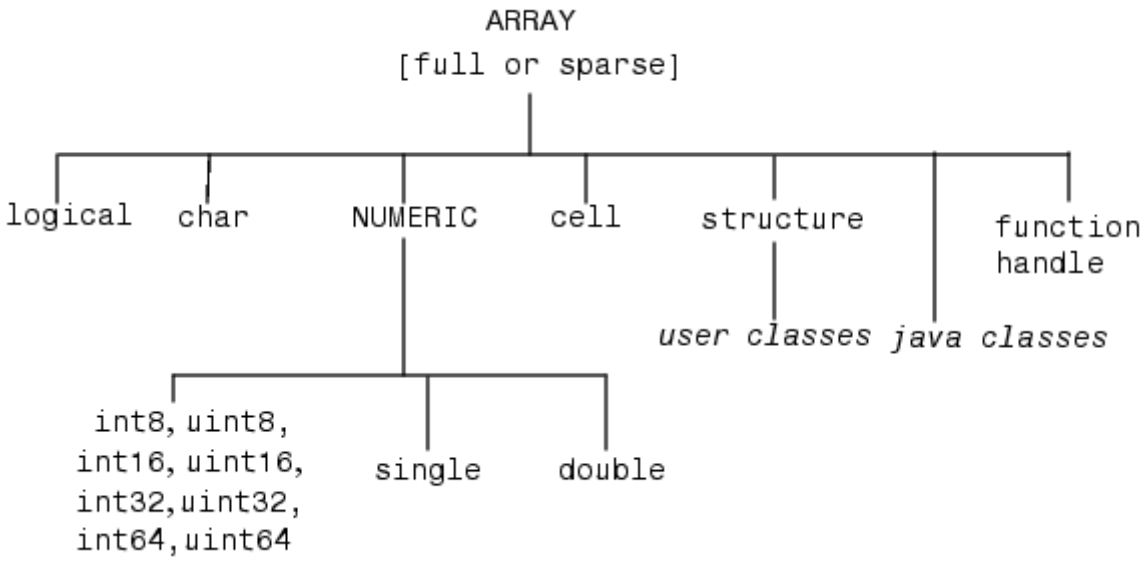
- **Function and operator overloading**. You can create methods that override existing MATLAB functions. When you call a function with a user-defined object as an argument, MATLAB first checks to see if there is a method defined for the object's class. If there is, MATLAB calls it, rather than the normal

MATLAB function.

- **Encapsulation of data and methods**. Object properties are not visible from the command line; you can access them only with class methods. This protects the object properties from operations that are not intended for the object's class.
- **Inheritance**. You can create class hierarchies of parent and child classes in which the child class inherits data fields and methods from the parent. A child class can inherit from one parent (*single inheritance*) or many parents (*multiple inheritance*). Inheritance can span one or more generations. Inheritance enables sharing common parent functions and enforcing common behavior amongst all child classes.
- **Aggregation**. You can create classes using *aggregation*, in which an object contains other objects. This is appropriate when an object type is part of another object type. For example, a savings account object might be a part of a financial portfolio object.

MATLAB Data Class Hierarchy

All MATLAB data types are designed to function as classes in object-oriented programming. The diagram below shows the fifteen fundamental data types (or classes) defined in MATLAB. You can add new data types to MATLAB by extending the class hierarchy.



The diagram shows a *user class* that inherits from the structure class. All classes that you create are structure based since this is the point in the class hierarchy where you can insert your own classes. (For more information about MATLAB data types, see the section on "Data Types.")

Creating Objects

You create an object by calling the class constructor and passing it the appropriate input arguments. In MATLAB, constructors have the same name as the class name. For example, the statement,

```
p = polynom([1 0 -2 -5]);
```

creates an object named `p` belonging to the class `polynom`. Once you have created a polynom object, you can operate on the object using methods that are defined for the `polynom` class. See Example -- A Polynomial Class for a description of the `polynom` class.

Invoking Methods on Objects

Class methods are M-file functions that take an object as one of the input arguments. The methods for a specific class must be placed in the class directory for that class (the `@class_name` directory). This is the first place that

MATLAB looks to find a class method.

The syntax for invoking a method on an object is similar to a function call. Generally, it looks like

```
[out1,out2,...] = method_name(object,arg1,arg2, ...);
```

For example, suppose a user-defined class called `polynom` has a `char` method defined for the class. This method converts a polynom object to a character string and returns the string. This statement calls the `char` method on the polynom object `p`.

```
s = char(p);
```

Using the `class` function, you can confirm that the returned value `s` is a character string.

```
class(s)ans =
     char
s
s =
     x^3-2*x-5
```

You can use the `methods` command to produce a list of all of the methods that are defined for a class.

Private Methods

Private methods can be called only by other methods of their class. You define private methods by placing the associated M-files in a `private` subdirectory of the `@class_name` directory. In the example,

```
@class_name/private/update_obj.m
```

the method `update_obj` has scope only within the `class_name` class. This means that `update_obj` can be called by any method that is defined in the `@class_name` directory, but it cannot be called from the MATLAB command line or by methods outside of the class directory, including parent methods.

Private methods and private functions differ in that private methods (in fact all methods) have an object as one of their input arguments and private functions do not. You can use private functions as helper functions, such as described in the next section.

Helper Functions

In designing a class, you may discover the need for functions that perform support tasks for the class, but do not directly operate on an object. These functions are called *helper functions*. A helper function can be a subfunction in a class method file or a private function. When determining which version of a particular function to call, MATLAB looks for these functions in the order listed above. For more information about the order in which MATLAB calls functions and methods, see How MATLAB Determines Which Method to Call.

Debugging Class Methods

You can use the MATLAB debugging commands with object methods in the same way that you use them with other M-files. The only difference is that you need to include the class directory name before the method name in the command call, as shown in this example using `dbstop`.

```
dbstop @polynom/char
```

While debugging a class method, you have access to all methods defined for the class, including inherited methods, private methods, and private functions.

Changing Class Definition

If you change the class definition, such as the number or names of fields in a class, you must issue a

```
clear classes
```

command to propagate the changes to your MATLAB session. This command also clears all objects from the workspace. See the `clear` command help entry for more information.

Setting Up Class Directories

The M-files defining the methods for a class are collected together in a directory referred to as the class directory. The directory name is formed with the class name preceded by the character @. For example, one of the examples used in this chapter is a class involving polynomials in a single variable. The name of the class, and the name of the class constructor, is `polynom`. The M-files defining a polynomial class would be located in directory with the name `@polynom`.

The class directories are subdirectories of directories on the MATLAB search path, but are not themselves on the path. For instance, the new `@polynom` directory could be a subdirectory of the MATLAB working directory or your own personal directory that has been added to the search path.

Adding the Class Directory to the MATLAB Path

After creating the class directory, you need to update the MATLAB path so that MATLAB can locate the class source files. The class directory should not be directly on the MATLAB path. Instead, you should add the parent directory to the MATLAB path. For example, if the `@polynom` class directory is located at

```
c:\my_classes\@polynom
```

you add the class directory to the MATLAB path with the `addpath` command

```
addpath c:\my_classes;
```

If you create a class directory with the same name as another class, MATLAB treats the two class directories as a single directory when locating class methods. For more information, see How MATLAB Determines Which Method to Call.

Data Structure

One of the first steps in the design of a new class is the choice of the data structure to be used by the class. Objects are stored in MATLAB structures. The fields of the structure, and the details of operations on the fields, are visible only within the methods for the class. The design of the appropriate data structure can affect the performance of the code.

Tips for C++ and Java Programmers

If you are accustomed to programming in other object-oriented languages, such as C++ or Java, you will find that the MATLAB programming language differs from these languages in some important ways:

- In MATLAB, method dispatching is not syntax based, as it is in C++ and Java. When the argument list contains objects of equal precedence, MATLAB uses the left-most object to select the method to call.
- In MATLAB, there is no equivalent to a destructor method. To remove an object from the workspace, use the `clear` function.
- Construction of MATLAB data types occurs at runtime rather than compile time. You register an object

as belonging to a class by calling the `class` function.

- When using inheritance in MATLAB, the inheritance relationship is established in the child class by creating the parent object, and then calling the `class` function. For more information on writing constructors for inheritance relationships, see Building on Other Classes.
- When using inheritance in MATLAB, the child object contains a parent object in a property with the name of the parent class.
- In MATLAB, there is no passing of variables by reference. When writing methods that update an object, you must pass back the updated object and use an assignment statement. For instance, this call to the `set` method updates the `name` field of the object `A` and returns the updated object.

```
A = set(A,'name','John Smith');
```

- In MATLAB, there is no equivalent to an abstract class.
- In MATLAB, there is no equivalent to the C++ scoping operator.
- In MATLAB, there is no virtual inheritance or virtual base classes.
- In MATLAB, there is no equivalent to C++ templates.

Designing User Classes in MATLAB

This section discusses how to approach the design of a class and describes the basic set of methods that should be included in a class.

The MATLAB Canonical Class

When you design a MATLAB class, you should include a standard set of methods that enable the class to behave in a consistent and logical way within the MATLAB environment. Depending on the nature of the class you are defining, you may not need to include all of these methods and you may include a number of other methods to realize the class's design goals.

This table lists the basic methods included in MATLAB classes.

Class Method	Description
class constructor	Creates an object of the class
`display`	Called whenever MATLAB displays the contents of an object (e.g., when an expression is entered without terminating with a semicolon)
`set` and `get`	Accesses class properties
`subsref` and `subsasgn`	Enables indexed reference and assignment for user objects
`end`	Supports `end` syntax in indexing expressions using an object; e.g., `A(1:end)`
`subsindex`	Supports using an object in indexing expressions
converters like `double` and `char`	Methods that convert an object to a MATLAB data type

The following sections discuss the implementation of each type of method, as well as providing references to examples used in this chapter.

The Class Constructor Method

The @ directory for a particular class must contain an M-file known as the *constructor* for that class. The name of the constructor is the same as the name of the directory (excluding the @ prefix and `.m` extension) that

defines the name of the class. The constructor creates the object by initializing the data structure and instantiating an object of the class.

Guidelines for Writing a Constructor

Class constructors must perform certain functions so that objects behave correctly in the MATLAB environment. In general, a class constructor must handle three possible combinations of input arguments:

- No input arguments
- An object of the same class as an input argument
- The input arguments used to create an object of the class (typically data of some kind)

No Input Arguments. If there are no input arguments, the constructor should create a default object. Since there are no inputs, you have no data from which to create the object, so you simply initialize the object's data structures with empty or default values, call the `class` function to instantiate the object, and return the object as the output argument. Support for this syntax is required for two reasons:

- When loading objects into the workspace, the `load` function calls the class constructor with no arguments.
- When creating arrays of objects, MATLAB calls the class constructor to add objects to the array.

Object Input Argument. If the first input argument in the argument list is an object of the same class, the constructor should simply return the object. Use the `isa` function to determine if an argument is a member of a class. See Overloading the + Operator for an example of a method that uses this constructor syntax.

Data Input Arguments. If the input arguments exist and are not objects of the same class, then the constructor creates the object using the input data. Of course, as in any function, you should perform proper argument checking in your constructor function. A typical approach is to use a `varargin` input argument and a `switch` statement to control program flow. This provides an easy way to accommodate the three cases: no inputs, object input, or the data inputs used to create an object.

It is in this part of the constructor that you assign values to the object's data structure, call the `class` function to instantiate the object, and return the object as the output argument. If necessary, place the object in an object hierarchy using the `superiorto` and `inferiorto` functions.

Using the class Function in Constructors

Within a constructor method, you use the `class` function to associate an object structure with a particular class. This is done using an internal class tag that is only accessible using the `class` and `isa` functions. For example, this call to the `class` function identifies the object `p` to be of type `polynom`.

```
p = class(p,'polynom');
```

Examples of Constructor Methods

See the following sections for examples of constructor methods:

- The Polynom Constructor Method
- The Asset Constructor Method
- The Stock Constructor Method
- The Portfolio Constructor Method

Identifying Objects Outside the Class Directory

The class and isa functions used in constructor methods can also be used outside of the class directory. The expression

```
isa(a,'class_name');
```

checks whether a is an object of the specified class. For example, if p is a polynom object, each of the following expressions is true.

```
isa(pi,'double');
isa('hello','char');
isa(p,'polynom');
```

Outside of the class directory, the class function takes only one argument (it is only within the constructor that class can have more than one argument).

The expression

```
class(a)
```

returns a string containing the class name of a. For example,

```
class(pi),
class('hello'),
class(p)
```

return

```
'double',
'char',
'polynom'
```

Use the whos function to see what objects are in the MATLAB workspace.

```
whos
  Name      Size          Bytes  Class
  p         1x1           156 polynom object
```

The display Method

MATLAB calls a method named display whenever an object is the result of a statement that is not terminated by a semicolon. For example, creating the variable a, which is a double, calls the MATLAB display method for doubles.

```
a = 5
a =
    5
```

You should define a display method so MATLAB can display values on the command line when referencing objects from your class. In many classes, display can simply print the variable name, and then use the char converter method to print the contents or value of the variable, since MATLAB displays output as strings. You must define the char method to convert the object's data to a character string.

Examples of display Methods

See the following sections for examples of display methods:

- The Polynom display Method
- The Asset display Method
- The Stock display Method
- The Portfolio display Method

Accessing Object Data

You need to write methods for your class that provide access to an object's data. Accessor methods can use a variety of approaches, but all methods that change object data always accept an object as an input argument and return a new object with the data changed. This is necessary because MATLAB does not support passing arguments by reference (i.e., pointers). Functions can change only their private, temporary copy of an object. Therefore, to change an existing object, you must create a new one, and then replace the old one.

The following sections provide more detail about implementation techniques for the `set`, `get`, `subsasgn`, and `subsref` methods.

The set and get Methods

The `set` and `get` methods provide a convenient way to access object data in certain cases. For example, suppose you have created a class that defines an arrow object that MATLAB can display on graphs (perhaps composed of existing MATLAB line and patch objects).

To produce a consistent interface, you could define `set` and `get` methods that operate on arrow objects the way the MATLAB `set` and `get` functions operate on built-in graphics objects. The `set` and `get` verbs convey what operations they perform, but insulate the user from the internals of the object.

Examples of set and get Methods

See the following sections for examples of `set` and `get` methods:

- The Asset get Method and The Asset set Method
- The Stock get Method and The Stock set Method

Property Name Methods

As an alternative to a general `set` method, you can write a method to handle the assignment of an individual property. The method should have the same name as the property name.

For example, if you defined a class that creates objects representing employee data, you might have a field in an employee object called `salary`. You could then define a method called `salary.m` that takes an employee object and a value as input arguments and returns the object with the specified value set.

Indexed Reference Using subsref and subsasgn

User classes implement new data types in MATLAB. It is useful to be able to access object data via an indexed reference, as is possible with the MATLAB built-in data types. For example, if `A` is an array of class `double`, `A(i)` returns the i^{th} element of `A`.

As the class designer, you can decide what an index reference to an object means. For example, suppose you define a class that creates polynomial objects and these objects contain the coefficients of the polynomial.

An indexed reference to a polynomial object,

```
p(3)
```

could return the value of the coefficient of x^3, the value of the polynomial at $x = 3$, or something different depending on the intended design.

You define the behavior of indexing for a particular class by creating two class methods - `subsref` and `subsasgn`. MATLAB calls these methods whenever a subscripted reference or assignment is made on an object from the class. If you do not define these methods for a class, indexing is undefined for objects of this class.

In general, the rules for indexing objects are the same as the rules for indexing structure arrays. For details, see Structures.

Handling Subscripted Reference

The use of a subscript or field designator with an object on the right-hand side of an assignment statement is known as a *subscripted reference*. MATLAB calls a method named `subsref` in these situations.

Object subscripted references can be of three forms - an array index, a cell array index, and a structure field name:

```
A(I)
A{I}
A.field
```

Each of these results in a call by MATLAB to the `subsref` method in the class directory. MATLAB passes two arguments to `subsref`.

```
B = subsref(A,S)
```

The first argument is the object being referenced. The second argument, `S`, is a structure array with two fields:

- `S.type` is a string containing `'()'`, `'{}'`, or `'.'` specifying the subscript type. The parentheses represent a numeric array; the curly braces, a cell array; and the dot, a structure array.
- `S.subs` is a cell array or string containing the actual subscripts. A colon used as a subscript is passed as the string `':'`.

For instance, the expression

```
A(1:2,:)
```

causes MATLAB to call `subsref(A,S)`, where `S` is a 1-by-1 structure with

```
S.type = '()'
S.subs = {1:2,':'}
```

Similarly, the expression

```
A{1:2}
```

uses

```
S.type ='{}'
S.subs = {1:2}
```

The expression

```
A.field
```

calls `subsref(A,S)` where

```
S.type = '.'
S.subs = 'field'
```

These simple calls are combined for more complicated subscripting expressions. In such cases, `length(S)` is the number of subscripting levels. For example,

```
A(1,2).name(3:4)
```

calls `subsref(A,S),` where `S` is a 3-by-1 structure array with the values:

```
S(1).type = '()'    S(2).type = '.'      S(3).type = '()'
S(1).subs = '{1,2}' S(2).subs = 'name'   S(3).subs = '{3:4}'
```

How to Write subsref

The `subsref` method must interpret the subscripting expressions passed in by MATLAB. A typical approach is to use the `switch` statement to determine the type of indexing used and to obtain the actual indices. The following three code fragments illustrate how to interpret the input arguments. In each case, the function must return the value `B`.

For an array index:

```
switch S.type
case '()'
    B = A(S.subs{:});
end
```

For a cell array:

```
switch S.type
case '{}'
    B = A(S.subs{:}); % A is a cell array
end
```

For a structure array:

```
switch S.type
case '.'
    switch S.subs
    case 'field1'
        B = A.field1;
    case 'field2'
        B = A.field2;
    end
end
```

Examples of the subsref Method

See the following sections for examples of the `subsref` method:

- The Polynom subsref Method

- The Asset subsref Method
- The Stock subsref Method
- The Portfolio subsref Method

Handling Subscripted Assignment

The use of a subscript or field designator with an object on the left-hand side of an assignment statement is known as a *subscripted assignment*. MATLAB calls a method named `subsasgn` in these situations. Object subscripted assignment can be of three forms - an array index, a cell array index, and a structure field name.

```
A(I) = B
A{I} = B
A.field = B
```

Each of these results in a call to `subsasgn` of the form

```
A = subsasgn(A,S,B)
```

The first argument, `A`, is the object being referenced. The second argument, `S`, has the same fields as those used with `subsref`. The third argument, `B`, is the new value.

Examples of the subsasgn Method

See the following sections for examples of the `subsasgn` method:

- The Asset subsasgn Method
- The Stock subsasgn Method

Object Indexing Within Methods

If a subscripted reference is made within a class method, MATLAB uses its built-in `subsref` function to access data within the method's own class. If the method accesses data from another class, MATLAB calls the overloaded `subsref` function in that class. The same holds true for subscripted assignment and `subsasgn`.

The following example shows a method, `testref`, that is defined in the class, `employee`. This method makes a reference to a field, `address`, in an object of its own class. For this, MATLAB uses the built-in `subsref` function. It also references the same field in another class, this time using the overloaded `subsref` of that class.

```
% ---- EMPLOYEE class method: testref.m ----
function testref(myclass,otherclass)

myclass.address                % use built-in subsref
otherclass.address             % use overloaded subsref
```

The example creates an `employee` object and a `company` object.

```
empl = employee('Johnson','Chicago');
comp = company('The MathWorks','Natick');
```

The `employee` class method, `testref`, is called. MATLAB uses an overloaded `subsref` only to access data outside of the method's own class.

```
testref(empl,comp)
ans =                          % built-in subsref was called
   Chicago
```

```
ans =                              % @company\subsref was calledExecuting @company\subs
   Natick
```

Defining end Indexing for an Object

When you use `end` in an object indexing expression, MATLAB calls the object's `end` class method. If you want to be able to use `end` in indexing expressions involving objects of your class, you must define an `end` method for your class.

The `end` method has the calling sequence

```
end(a,k,n)
```

where `a` is the user object, `k` is the index in the expression where the `end` syntax is used, and `n` is the total number of indices in the expression.

For example, consider the expression

```
A(end-1,:)
```

MATLAB calls the `end` method defined for the object `A` using the arguments

```
end(A,1,2)
```

That is, the `end` statement occurs in the first index element and there are two index elements. The class method for `end` must then return the index value for the last element of the first dimension. When you implement the `end` method for your class, you must ensure it returns a value appropriate for the object.

Indexing an Object with Another Object

When MATLAB encounters an object as an index, it calls the `subsindex` method defined for the object. For example, suppose you have an object `a` and you want to use this object to index into another object `b`.

```
c = b(a);
```

A `subsindex` method might do something as simple as convert the object to double format to be used as an index, as shown in this sample code.

```
function d = subsindex(a)
%SUBSINDEX
% convert the object a to double format to be used
% as an index in an indexing expression
d = double(a);
```

`subsindex` values are 0-based, not 1-based.

Converter Methods

A converter method is a class method that has the same name as another class, such as `char` or `double`. Converter methods accept an object of one class as input and return an object of another class. Converters enable you to:

- Use methods defined for another class
- Ensure that expressions involving objects of mixed class types execute properly

A converter function call is of the form

```
b = class_name(a)
```

where `a` is an object of a class other than *`class_name`*. In this case, MATLAB looks for a method called *`class_name`* in the class directory for object `a`. If the input object is already of type *`class_name`*, then MATLAB calls the constructor, which just returns the input argument.

Examples of Converter Methods

See the following sections for examples of converter methods:

- The Polynom to Double Converter
- The Polynom to Char Converter

Overloading Operators and Functions

In many cases, you may want to change the behavior of the MATLAB operators and functions for cases when the arguments are objects. You can accomplish this by overloading the relevant functions. Overloading enables a function to handle different types and numbers of input arguments and perform whatever operation is appropriate for the highest-precedence object. See Object Precedence for more information on object precedence.

Overloading Operators

Each built-in MATLAB operator has an associated function name (e.g., the + operator has an associated `plus.m` function). You can overload any operator by creating an M-file with the appropriate name in the class directory. For example, if either `p` or `q` is an object of type *`class_name`*, the expression

```
p + q
```

generates a call to a function `@`*`class_name`*`/plus.m`, if it exists. If `p` and `q` are both objects of different classes, then MATLAB applies the rules of precedence to determine which method to use.

Examples of Overloaded Operators

See the following sections for examples of overloaded operators:

- Overloading the + Operator
- Overloading the - Operator
- Overloading the * Operator

The following table lists the function names for most of the MATLAB operators.

Operation	M-File	Description
`a + b`	`plus(a,b)`	Binary addition
`a - b`	`minus(a,b)`	Binary subtraction
`-a`	`uminus(a)`	Unary minus
`+a`	`uplus(a)`	Unary plus
`a.*b`	`times(a,b)`	Element-wise multiplication

a*b	mtimes(a,b)	Matrix multiplication
a./b	rdivide(a,b)	Right element-wise division
a.\b	ldivide(a,b)	Left element-wise division
a/b	mrdivide(a,b)	Matrix right division
a\b	mldivide(a,b)	Matrix left division
a.^b	power(a,b)	Element-wise power
a^b	mpower(a,b)	Matrix power
a < b	lt(a,b)	Less than
a > b	gt(a,b)	Greater than
a <= b	le(a,b)	Less than or equal to
a >= b	ge(a,b)	Greater than or equal to
a ~= b	ne(a,b)	Not equal to
a == b	eq(a,b)	Equality
a & b	and(a,b)	Logical AND
a | b	or(a,b)	Logical OR
~a	not(a)	Logical NOT
a:d:b a:b	colon(a,d,b) colon(a,b)	Colon operator
a'	ctranspose(a)	Complex conjugate transpose
a.'	transpose(a)	Matrix transpose
command window output	display(a)	Display method
[a b]	horzcat(a,b,...)	Horizontal concatenation
[a; b]	vertcat(a,b,...)	Vertical concatenation
a(s1,s2,...sn)	subsref(a,s)	Subscripted reference
a(s1,...,sn) = b	subsasgn(a,s,b)	Subscripted assignment
b(a)	subsindex(a)	Subscript index

Overloading Functions

You can overload any function by creating a function of the same name in the class directory. When a function is invoked on an object, MATLAB always looks in the class directory before any other location on the search path. To overload the `plot` function for a class of objects, for example, simply place your version of `plot.m` in the appropriate class directory.

Examples of Overloaded Functions

See the following sections for examples of overloaded functions:

- Overloading Functions for the Polynom Class
- The Portfolio pie3 Method

Example -- A Polynomial Class

This example implements a MATLAB data type for polynomials by defining a new class called polynom. The class definition specifies a structure for data storage and defines a directory (`@polynom`) of methods that operate on polynom objects.

Polynom Data Structure

The polynom class represents a polynomial with a row vector containing the coefficients of powers of the variable, in decreasing order. Therefore, a polynom object `p` is a structure with a single field, `p.c`, containing the coefficients. This field is accessible only within the methods in the `@polynom` directory.

Polynom Methods

To create a class that is well behaved within the MATLAB environment and provides useful functionality for a polynomial data type, the polynom class implements the following methods:

- A constructor method `polynom.m`
- A polynom to double converter
- A polynom to char converter
- A `display` method
- A `subsref` method
- Overloaded `+`, `-`, and `*` operators
- Overloaded `roots`, `polyval`, `plot`, and `diff` functions

The Polynom Constructor Method

Here is the polynom class constructor, `@polynom/polynom.m`.

```
function p = polynom(a)
%POLYNOM Polynomial class constructor.
%   p = POLYNOM(v) creates a polynomial object from the vector v,
%   containing the coefficients of descending powers of x.
if nargin == 0
   p.c = [];
   p = class(p,'polynom');
elseif isa(a,'polynom')
   p = a;
else
   p.c = a(:).';
   p = class(p,'polynom');
end
```

Constructor Calling Syntax

You can call the polynom constructor method with one of three different arguments:

- No Input Argument - If you call the constructor function with no arguments, it returns a polynom object with empty fields.
- Input Argument is an Object - If you call the constructor function with an input argument that is already a polynom object, MATLAB returns the input argument. The `isa` function (pronounced "is a") checks for this situation.
- Input Argument is a coefficient vector - If the input argument is a variable that is not a polynom object, reshape it to be a row vector and assign it to the `.c` field of the object's structure. The `class` function

creates the `polynom` object, which is then returned by the constructor.

An example use of the `polynom` constructor is the statement

```
p = polynom([1 0 -2 -5])
```

This creates a polynomial with the specified coefficients.

Converter Methods for the Polynom Class

A converter method converts an object of one class to an object of another class. Two of the most important converter methods contained in MATLAB classes are `double` and `char`. Conversion to `double` produces the MATLAB traditional matrix, although this may not be appropriate for some classes. Conversion to `char` is useful for producing printed output.

The Polynom to Double Converter

The double converter method for the polynom class is a very simple M-file, `@polynom/double.m`, which merely retrieves the coefficient vector.

```
function c = double(p)
% POLYNOM/DOUBLE  Convert polynom object to coefficient vector.
%   c = DOUBLE(p) converts a polynomial object to the vector c
%   containing the coefficients of descending powers of x.
c = p.c;
```

On the object `p`,

```
p = polynom([1 0 -2 -5])
```

the statement

```
double(p)
```

returns

```
ans =
    1     0    -2    -5
```

The Polynom to Char Converter

The converter to char is a key method because it produces a character string involving the powers of an independent variable, `x`. Therefore, once you have specified `x`, the string returned is a syntactically correct MATLAB expression, which you can then evaluate.

Here is `@polynom/char.m`.

```
function s = char(p)
% POLYNOM/CHAR
% CHAR(p) is the string representation of p.c
if all(p.c == 0)
    s = '0';
else
    d = length(p.c) - 1;
    s = [];
    for a = p.c;
        if a ~= 0;
```

```
                if ~isempty(s)
                     if a > 0
                         s = [s ' + '];
                     else
                         s = [s ' - '];
                         a = -a;
                     end
                end
                if a ~= 1 | d == 0
                     s = [s num2str(a)];
                     if d > 0
                         s = [s '*'];
                     end
                end
                if d >= 2
                     s = [s 'x^' int2str(d)];
                elseif d == 1
                     s = [s 'x'];
                end
            end
            d = d - 1;
        end
    end
```

Evaluating the Output

If you create the polynom object `p`

```
p = polynom([1 0 -2 -5]);
```

and then call the `char` method on `p`

```
char(p)
```

MATLAB produces the result

```
ans =
    x^3 - 2*x - 5
```

The value returned by `char` is a string that you can pass to `eval` once you have defined a scalar value for `x`. For example,

```
x = 3;

eval(char(p))
ans =
     16
```

See The Polynom subsref Method for a better method to evaluate the polynomial.

The Polynom display Method

Here is `@polynom/display.m`. This method relies on the `char` method to produce a string representation of the polynomial, which is then displayed on the screen. This method produces output that is the same as standard MATLAB output. That is, the variable name is displayed followed by an equal sign, then a blank line, then a new line with the value.

```
function display(p)
% POLYNOM/DISPLAY Command window display of a polynom
disp(' ');
```

```
    disp([inputname(1),' = '])
    disp(' ');
    disp(['    ' char(p)])
    disp(' ');
```

The statement

```
    p = polynom([1 0 -2 -5])
```

creates a polynom object. Since the statement is not terminated with a semicolon, the resulting output is

```
    p =
        x^3 - 2*x - 5
```

The Polynom subsref Method

Suppose the design of the polynom class specifies that a subscripted reference to a `polynom` object causes the polynomial to be evaluated with the value of the independent variable equal to the subscript. That is, for a polynom object `p`,

```
    p = polynom([1 0 -2 -5]);
```

the following subscripted expression returns the value of the polynomial at `x = 3` and `x = 4`.

```
    p([3 4])
    ans =
        16   51
```

subsref Implementation Details

This implementation takes advantage of the `char` method already defined in the polynom class to produce an expression that can then be evaluated.

```
    function b = subsref(a,s)
    % SUBSREF
    switch s.type
    case '()'
        ind = s.subs{:};
        for k = 1:length(ind)
            b(k) = eval(strrep(char(a),'x',num2str(ind(k))));
        end
    otherwise
       error('Specify value for x as p(x)')
    end
```

Once the polynomial expression has been generated by the `char` method, the `strrep` function is used to swap the passed in value for the character `x`. The `eval` function then evaluates the expression and returns the value in the output argument.

Overloading Arithmetic Operators for polynom

Several arithmetic operations are meaningful on polynomials and should be implemented for the polynom class. When overloading arithmetic operators, keep in mind what data types you want to operate on. In this section, the `plus`, `minus`, and `mtimes` methods are defined for the polynom class to handle addition, subtraction, and multiplication on polynom/polynom and polynom/double combinations of operands.

Overloading the + Operator

If either `p` or `q` is a polynom, the expression

```
p + q
```

generates a call to a function `@polynom/plus.m`, if it exists (unless `p` or `q` is an object of a higher precedence, as described in Object Precedence).

The following M-file redefines the + operator for the polynom class.

```
function r = plus(p,q)
% POLYNOM/PLUS  Implement p + q for polynoms.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] + [zeros(1,-k) q.c]);
```

The function first makes sure that both input arguments are polynomials. This ensures that expressions such as

```
p + 1
```

that involve both a polynom and a double, work correctly. The function then accesses the two coefficient vectors and, if necessary, pads one of them with zeros to make them the same length. The actual addition is simply the vector sum of the two coefficient vectors. Finally, the function calls the `polynom` constructor a third time to create the properly typed result.

Overloading the - Operator

You can implement the overloaded minus operator (-) using the same approach as the plus (+) operator. MATLAB calls `@polynom/minus.m` to compute `p-q`.

```
function r = minus(p,q)
% POLYNOM/MINUS Implement p - q for polynoms.
p = polynom(p);
q = polynom(q);
k = length(q.c) - length(p.c);
r = polynom([zeros(1,k) p.c] - [zeros(1,-k) q.c]);
```

Overloading the * Operator

MATLAB calls the method `@polynom/mtimes.m` to compute the product `p*q`. The letter `m` at the beginning of the function name comes from the fact that it is overloading the MATLAB *matrix* multiplication. Multiplication of two polynomials is simply the convolution of their coefficient vectors.

```
function r = mtimes(p,q)
% POLYNOM/MTIMES   Implement p * q for polynoms.
p = polynom(p);
q = polynom(q);
r = polynom(conv(p.c,q.c));
```

Using the Overloaded Operators

Given the polynom object

```
p = polynom([1 0 -2 -5])
```

MATLAB calls these two functions `@polynom/plus.m` and `@polynom/mtimes.m` when you issue the statements

```
q = p+1
r = p*q
```

to produce

```
q =
     x^3 - 2*x - 4

r =
    x^6 - 4*x^4 - 9*x^3 + 4*x^2 + 18*x + 20
```

Overloading Functions for the Polynom Class

MATLAB already has several functions for working with polynomials represented by coefficient vectors. They should be overloaded to also work with the new polynom object. In many cases, the overloading methods can simply apply the original function to the coefficient field.

Overloading roots for the Polynom Class

The method `@polynom/roots.m` finds the roots of polynom objects.

```
function r = roots(p)
% POLYNOM/ROOTS.  ROOTS(p) is a vector containing the roots of p.
r = roots(p.c);
```

The statement

```
roots(p)
```

results in

```
ans =
    2.0946
    -1.0473 + 1.1359i
    -1.0473 - 1.1359i
```

Overloading polyval for the Polynom Class

The function `polyval` evaluates a polynomial at a given set of points. `@polynom/polyval.m` uses nested multiplication, or Horner's method to reduce the number of multiplication operations used to compute the various powers of x.

```
function y = polyval(p,x)
% POLYNOM/POLYVAL  POLYVAL(p,x) evaluates p at the points x.
y = 0;
for a = p.c
   y = y.*x + a;
end
```

Overloading plot for the Polynom Class

The overloaded `plot` function uses both `root` and `polyval`. The function selects the domain of the independent variable to be slightly larger than an interval containing all real roots. Then `polyval` is used to evaluate the polynomial at a few hundred points in the domain.

```
function plot(p)
% POLYNOM/PLOT  PLOT(p) plots the polynom p.
r = max(abs(roots(p)));
x = (-1.1:0.01:1.1)*r;
y = polyval(p,x);
plot(x,y);
title(char(p))
grid on
```

Overloading diff for the Polynom Class

`The method @polynom/diff.m` differentiates a polynomial by reducing the degree by 1 and multiplying each coefficient by its original degree.

```
function q = diff(p)
% POLYNOM/DIFF  DIFF(p) is the derivative of the polynom p.
c = p.c;
d = length(c) - 1;  % degree
q = polynom(p.c(1:d).*(d:-1:1));
```

Listing Class Methods

The function call

```
methods('class_name')
```

or its command form

```
methods class_name
```

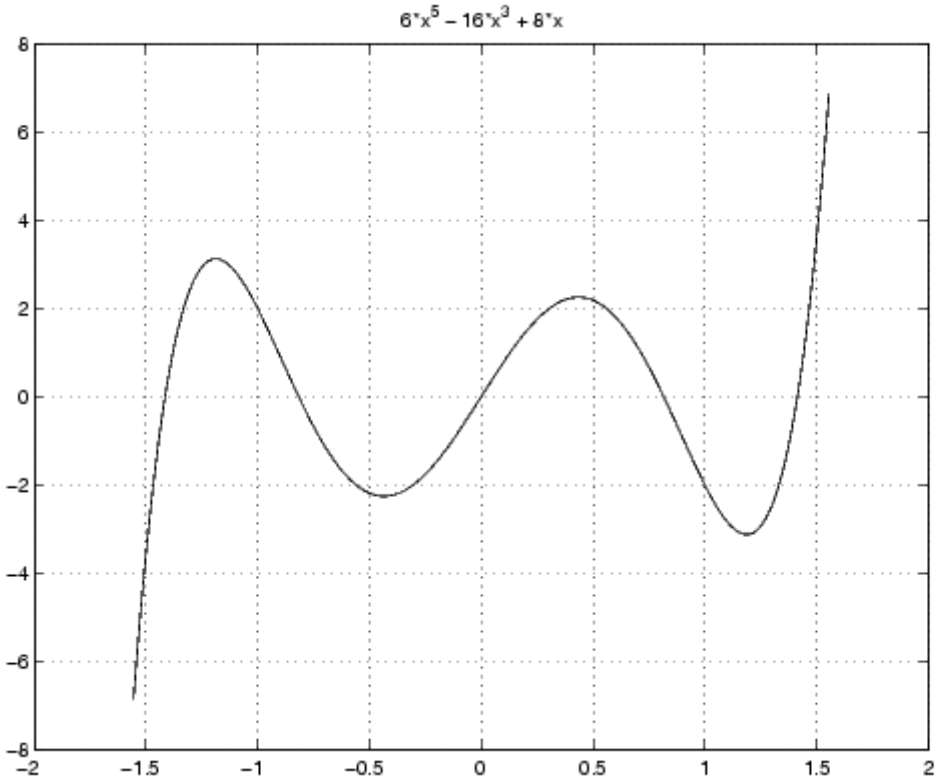
shows all the methods available for a particular class. For the `polynom` example, the output is

```
methods polynom
Methods for class polynom:
```

char	display	minus	plot	polynom	roots
diff	double	mtimes	plus	polyval	subsref

Plotting the two polynom objects `x` and `p` calls most of these methods.

```
x = polynom([1 0]);
p = polynom([1 0 -2 -5]);
plot(diff(p*p + 10*p + 20*x) - 20)
```



Building on Other Classes

A MATLAB object can *inherit* properties and behavior from another MATLAB object. When one object (the child) inherits from another (the parent), the child object includes all the fields of the parent object and can call the parent's methods. The parent methods can access those fields that a child object inherited from the parent class, but not fields new to the child class.

Inheritance is a key feature of object-oriented programming. It makes it easy to reuse code by allowing child objects to take advantage of code that exists for parent objects. Inheritance enables a child object to behave exactly like a parent object, which facilitates the development of related classes that behave similarly, but are implemented differently.

There are two kinds of inheritance:

- Simple inheritance, in which a child object inherits characteristics from one parent class.
- Multiple inheritance, in which a child object inherits characteristics from more than one parent class.

This section also discusses a related topic, aggregation. Aggregation allows one object to contain another object as one of its fields.

Saving and Loading Objects

You can use the MATLAB `save` and `load` commands to save and retrieve user-defined objects to and from `.mat` files, just like any other variables.

When you load objects, MATLAB calls the object's class constructor to register the object in the workspace. The constructor function for the object class you are loading must be able to be called with no input arguments and return a default object. See Guidelines for Writing a Constructor for more information.

Modifying Objects During Save or Load

When you issue a `save` or `load` command on objects, MATLAB looks for class methods called `saveobj` and `loadobj` in the class directory. You can overload these methods to modify the object before the save or load operation. For example, you could define a `saveobj` method that saves related data along with the object or you could write a `loadobj` method that updates objects to a newer version when this type of object is loaded into the MATLAB workspace.

Object Precedence

Object precedence is a means to resolve the question of which of possibly many versions of an operator or function to call in a given situation. Object precedence enables you to control the behavior of expressions containing different classes of objects. For example, consider the expression

```
objectA + objectB
```

Ordinarily, MATLAB assumes that the objects have equal precedence and calls the method associated with the leftmost object. However, there are two exceptions:

- User-defined classes have precedence over MATLAB built-in classes.
- User-defined classes can specify their relative precedence with respect to other user-defined classes using the `inferiorto` and `superiorto` functions.

For example, in the section Example -- A Polynomial Class the polynom class defines a `plus` method that enables addition of polynom objects. Given the polynom object `p`

```
p = polynom([1 0 -2 -5])
p =
    x^3-2*x-5
```

The expression,

```
1 + p
ans =
    x^3-2*x-4
```

calls the polynom `plus` method (which converts the double, 1, to a polynom object, and then adds it to `p`). The user-defined polynom class has precedence over the MATLAB double class.

Specifying Precedence of User-Defined Classes

You can specify the relative precedence of user-defined classes by calling the `inferiorto` or `superiorto` function in the class constructor.

The `inferiorto` function places a class below other classes in the precedence hierarchy. The calling syntax for the `inferiorto` function is

```
inferiorto('class1','class2',...)
```

You can specify multiple classes in the argument list, placing the class below many other classes in the hierarchy.

Similarly, the `superiorto` function places a class above other classes in the precedence hierarchy. The calling syntax for the `superiorto` function is

```
superiorto('class1','class2',...)
```

Location in the Hierarchy

If *`objectA`* is above *`objectB`* in the precedence hierarchy, then the expression

```
objectA + objectB
```

calls `@classA/plus.m`. Conversely, if *`objectB`* is above *`objectA`* in the precedence hierarchy, then MATLAB calls `@classB/plus.m`.

See How MATLAB Determines Which Method to Call for related information.

How MATLAB Determines Which Method to Call

In MATLAB, functions exist in directories in the computer's file system. A directory may contain many functions (M-files). Function names are unique only within a single directory (e.g., more than one directory may contain a function called `pie3`). When you type a function name on the command line, MATLAB must search all the directories it is aware of to determine which function to call. This list of directories is called the *MATLAB path*.

When looking for a function, MATLAB searches the directories in the order they are listed in the path, and calls the first function whose name matches the name of the specified function.

If you write an M-file called `pie3.m` and put it in a directory that is searched before the `specgraph` directory that contains the MATLAB `pie3` function, then MATLAB uses your `pie3` function instead (note that this is not true for built-in functions like `plot`, which are always found first).

Object-oriented programming allows you to have many methods (MATLAB functions located in class directories) with the same name and enables MATLAB to determine which method to use based on the type or class of the variables passed to the function. For example, if `p` is a portfolio object, then

```
pie3(p)
```

calls `@portfolio/pie3.m` because the argument is a portfolio object.

Selecting a Method

When you call a method for which there are multiple versions with the same name, MATLAB determines the method to call by:

- Looking at the classes of the objects in the argument list to determine which argument has the highest object precedence; the class of this object controls the method selection and is called the *dispatch type*.
- Applying the *function precedence order* to determine which of possibly several implementations of a method to call. This order is determined by the location and type of function.

Determining the Dispatch Type

MATLAB first determines which argument controls the method selection. The class type of this argument then determines the class in which MATLAB searches for the method. The controlling argument is either:

- The argument with the highest precedence, or
- The leftmost of arguments having equal precedence

User-defined objects take precedence over the MATLAB built-in classes such as `double` or `char`. You can set the relative precedence of user-defined objects with the `inferiorto` and `superiorto` functions, as described in Object Precedence.

MATLAB searches for functions by name. When you call a function, MATLAB knows the name, number of arguments, and the type of each argument. MATLAB uses the dispatch type to choose among multiple functions of the same name, but does not consider the number of arguments.

Function Precedence Order

The function precedence order determines the precedence of one function over another based on the type of function and its location on the MATLAB path. From the perspective of method selection, MATLAB contains two types of functions: those built into MATLAB, and those written as M-files. MATLAB treats these types differently when determining the function precedence order.

MATLAB selects the correct function for a given context by applying the following function precedence rules, in the order given.

For built-in functions:

1. Overloaded Methods

   If there is a method in the class directory of the dispatching argument that has the same name as a MATLAB built-in function, then this method is called instead of the built-in function.

2. Nonoverloaded MATLAB Functions

   If there is no overloaded method, then the MATLAB built-in function is called.

MATLAB built-in functions take precedence over both subfunctions and private functions. Therefore, subfunctions or private functions with the same name as MATLAB built-in functions can never be called.

For nonbuilt-in functions:

1. Subfunctions

   Subfunctions take precedence over all other M-file functions and overloaded methods that are on the path and have the same name. Even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the subfunction and ignores the overloaded method.

2. Private Functions

   Private functions are called if there is no subfunction of the same name within the current scope. As with subfunctions, even if the function is called with an argument of type matching that of an overloaded method, MATLAB uses the private function and ignores the overloaded method.

3. Class Constructor Functions

   Constructor functions (functions having names that are the same as the @ directory, for example `@polynom/polynom.m`) take precedence over other MATLAB functions. Therefore, if you create an M-file called `polynom.m` and put it on your path before the constructor `@polynom/polynom.m` version, MATLAB will always call the constructor version.

4. Overloaded Methods

   MATLAB calls an overloaded method if it is not masked by a subfunction or private function.

5. Current Directory

   A function in the current working directory is selected before one elsewhere on the path.

6. Elsewhere On Path

   Finally, a function anywhere else on the path is selected.

Selecting Methods from Multiple Directories

There may be a number of directories on the path that contain methods with the same name. MATLAB stops searching when it finds the first implementation of the method on the path, regardless of the implementation type (MEX-file, P-code, M-file).

Selecting Methods from Multiple Implementation Types

There are four file precedence types. MATLAB uses file precedence to select between identically named functions in the same directory. The order of precedence for file types is:

1. MEX-files
2. MDL-file (Simulink model)
3. P-code
4. M-file

For example, if MATLAB finds a P-code and an M-file version of a method in a class directory, then the P-code version is used. It is, therefore, important to regenerate the P-code version whenever you edit the M-file.

Querying Which Method MATLAB Will Call

You can determine which method MATLAB will call using the `which` command. For example,

```
which pie3
your_matlab_path/toolbox/matlab/specgraph/pie3.m
```

However, if `p` is a portfolio object,

```
which pie3(p)
dir_on_your_path/@portfolio/pie3.m % portfolio method
```

The `which` command determines which version of `pie3` MATLAB will call if you passed a portfolio object as the input argument. To see a list of all versions of a particular function that are on your MATLAB path, use the `-all` option. See the `which` reference page for more information on this command.

---

disp

Display text or array

Syntax

```
disp(X)
```

Description

`disp(X)` displays an array, without printing the array name. If `X` contains a text string, the string is displayed.

Another way to display an array on the screen is to type its name, but this prints a leading `"X =,"` which is not always desirable.

Note that `disp` does not display empty arrays.

Examples

One use of `disp` in an M-file is to display a matrix with column labels:

```
disp('         Corn         Oats         Hay')
disp(rand(5,3))
```

which results in

```
    Corn         Oats         Hay
  0.2113       0.8474       0.2749
  0.0820       0.4524       0.8807
  0.7599       0.8075       0.6538
  0.0087       0.4832       0.4899
  0.8096       0.6135       0.7741
```

See Also

`format`, `int2str`, `num2str`, `rats`, `sprintf`

---

subsref

Overloaded method for `A(I)`, `A{I}` and `A.field`

Syntax

```
B = subsref(A,S)
```

Description

`B = subsref(A,S)` is called for the syntax `A(i)`, `A{i}`, or `A.i` when `A` is an object. `S` is a structure array with the fields

- `type`: A string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs`: A cell array or string containing the actual subscripts.

Remarks

`subsref` is designed to be used by the MATLAB interpreter to handle indexed references to objects. Calling

`subsref` directly as a function is not recommended. If you do use `subsref` in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.

Examples

The syntax `A(1:2,:)` calls `subsref(A,S)` where `S` is a 1-by-1 structure with `S.type='()'` and `S.subs={1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

The syntax `A{1:2}` calls `subsref(A,S)` where `S.type='{}'` and `S.subs={1:2}`.

The syntax `A.field` calls `subsref(A,S)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)` calls `subsref(A,S)` where `S` is a 3-by-1 structure array with the following values:

```
S(1).type='()'   S(2).type='.'     S(3).type='()'
S(1).subs={1,2}  S(2).subs='name'  S(3).subs={3:5}
```

See Also

`subsasgn`

See Handling Subscripted Reference for more information about overloaded methods and `subsref`.

---

subsasgn

Overloaded method for `A(I)=B`, `A{I}=B`, and `A.field=B`

Syntax

```
A = subsasgn(A,S,B)
```

Description

`A = subsasgn(A,S,B)` is called for the syntax `A(i)=B`, `A{i}=B`, or `A.i=B` when `A` is an object. `S` is a structure array with the fields

- `type`: A string containing `'()'`, `'{}'`, or `'.'`, where `'()'` specifies integer subscripts, `'{}'` specifies cell array subscripts, and `'.'` specifies subscripted structure fields.
- `subs`: A cell array or string containing the actual subscripts.

Remarks

`subsasgn` is designed to be used by the MATLAB interpreter to handle indexed assignments to objects. Calling `subsasgn` directly as a function is not recommended. If you do use `subsasgn` in this way, it conforms to the formal MATLAB dispatching rules and can yield unexpected results.

Examples

The syntax `A(1:2,:)=B` calls `A=subsasgn(A,S,B)` where `S` is a 1-by-1 structure with `S.type='()'` and

`S.subs = {1:2,':'}`. A colon used as a subscript is passed as the string `':'`.

The syntax `A{1:2}=B` calls `A=subsasgn(A,S,B)` where `S.type='{}'`.

The syntax `A.field=B` calls `subsasgn(A,S,B)` where `S.type='.'` and `S.subs='field'`.

These simple calls are combined in a straightforward way for more complicated subscripting expressions. In such cases `length(S)` is the number of subscripting levels. For instance, `A(1,2).name(3:5)=B` calls `A=subsasgn(A,S,B)` where `S` is a 3-by-1 structure array with the following values:

```
S(1).type='()'   S(2).type='.'     S(3).type='()'
S(1).subs={1,2}  S(2).subs='name'  S(3).subs={3:5}
```

See Also

`subsref`

See Handling Subscripted Assignment for more information about overloaded methods and `subsasgn`.

---

end

Terminate `for`, `while`, `switch`, `try`, and `if` statements or indicate last index

Syntax

```
while expression % (or if, for, or try)
     statements
end
B = A(index:end,index)
```

Description

`end` is used to terminate `for`, `while`, `switch`, `try`, and `if` statements. Without an `end` statement, `for`, `while`, `switch`, `try`, and `if` wait for further input. Each `end` is paired with the closest previous unpaired `for`, `while`, `switch`, `try`, or `if` and serves to delimit its scope.

The `end` command also serves as the last index in an indexing expression. In that context, `end = (size(x,k))` when used as part of the `k`th index. Examples of this use are `X(3:end)` and `X(1,1:2:end-1)`. When using `end` to grow an array, as in `X(end+1)=5`, make sure `X` exists first.

You can overload the `end` statement for a user object by defining an `end` method for the object. The `end` method should have the calling sequence `end(obj,k,n)`, where `obj` is the user object, `k` is the index in the expression where the `end` syntax is used, and `n` is the total number of indices in the expression. For example, consider the expression

```
A(end-1,:)
```

MATLAB will call the `end` method defined for `A` using the syntax

```
end(A,1,2)
```

Examples

This example shows `end` used with the `for` and `if` statements.

```
for k = 1:n
    if a(k) == 0
        a(k) = a(k) + 2;
    end
end
```

In this example, `end` is used in an indexing expression.

```
A = magic(5)

A =

    17    24     1     8    15
    23     5     7    14    16
     4     6    13    20    22
    10    12    19    21     3
    11    18    25     2     9

B = A(end,2:end)

B =

    18    25     2     9
```

See Also

`break`, `for`, `if`, `return`, `switch`, `try`, `while`

---

subsindex

Overloaded method for `X(A)`

Syntax

```
ind = subsindex(A)
```

Description

`ind = subsindex(A)` is called for the syntax `'X(A)'` when `A` is an object. `subsindex` must return the value of the object as a zero-based integer index. (`ind` must contain integer values in the range `0` to `prod(size(X))-1`.) `subsindex` is called by the default `subsref` and `subsasgn` functions, and you can call it if you overload these functions.

See Also

`subsasgn`, `subsref`

---

double

Convert to double precision

Syntax

```
double(X)
```

Description

`double(x)` returns the double-precision value for `X`. If `X` is already a double-precision array, `double` has no effect.

Remarks

`double` is called for the expressions in `for`, `if`, and `while` loops if the expression isn't already double-precision. `double` should be overloaded for any object when it makes sense to convert it to a double-precision value.

---

char

Create character array (string)

Syntax

```
S = char(X)
S = char(C)
S = char(t1,t2,t3...)
```

Description

`S = char(X)` converts the array `X` that contains positive integers representing character codes into a MATLAB character array (the first 127 codes are ASCII). The actual characters displayed depend on the character set encoding for a given font. The result for any elements of `X` outside the range from 0 to 65535 is not defined (and can vary from platform to platform). Use `double` to convert a character array into its numeric codes.

`S = char(C),` when `C` is a cell array of strings, places each element of `C` into the rows of the character array `s`. Use `cellstr` to convert back.

`S = char(t1,t2,t3,..)` forms the character array `S` containing the text strings `T1`,`T2`,`T3`,... as rows, automatically padding each string with blanks to form a valid matrix. Each text parameter, `T`*`i`*, can itself be a character array. This allows the creation of arbitrarily large character arrays. Empty strings are significant.

Remarks

Ordinarily, the elements of `A` are integers in the range 32:127, which are the printable ASCII characters, or in the range 0:255, which are all 8-bit values. For noninteger values, or values outside the range 0:255, the characters printed are determined by `fix(rem(A,256))`.

Examples

To print a 3-by-32 display of the printable ASCII characters,

```
ascii = char(reshape(32:127,32,3)')
ascii =
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _
' a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~
```

See Also

cellstr, double, get, set, strings, strvcat, text