Welcome back to **CME 292**
Advanced MATLAB for Scientific Computing

WINTER 2023

Stanford University

Advanced Programming Techniques

CME 292 LECTURE 6

1/26/2023

Outline

Object Oriented Programming

- OOP fundamentals and features
- Class components
- Implementing operators
- Value vs handle classes
- Event and listener

Efficient Code Writing

- Publishing coding
- Code debugger and analyzer
- Profiler and benchmark
- Code performance and memory management
- Parallel processing

Object Oriented Programming

Advanced Programming Techniques

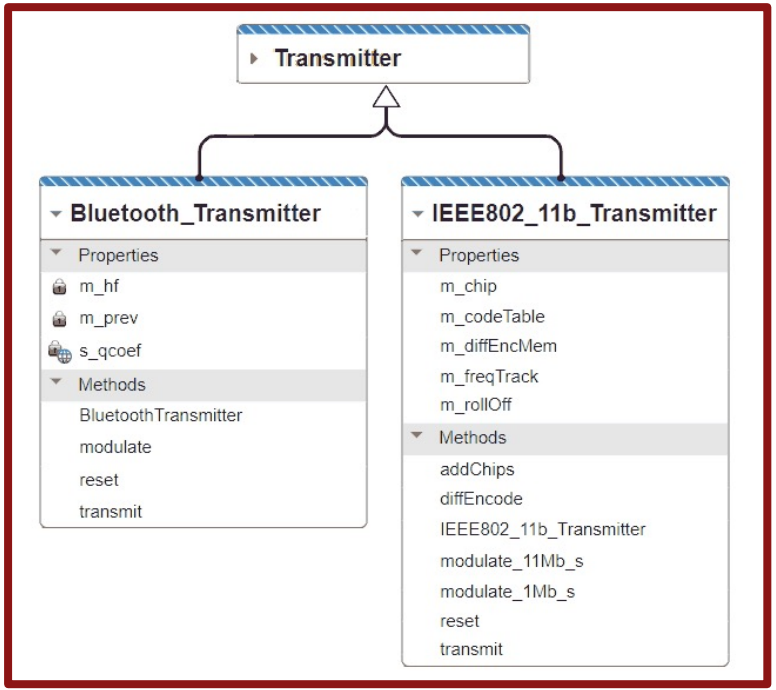


Image from MathWorks

OOP Fundamentals

OOP is a programming paradigm organized around objects equipped with data fields and associated methods.

- Solver's configuration parameters (properties) are grouped with its functions (methods) into a single definition, or **class**.
- Data (state) and methods (behavior) are associated via objects
- **Objects** used to interact with each other
- An object is an instance of a class.
- Languages: C++, Objective-C, Java, C#, Perl, Python, Ruby, PHP, etc.

OOP enables a level of modularity and abstraction not generally available in procedural languages:

- Increased code understanding
- Code maintenance
- Code expansion/evolution

OOP Features

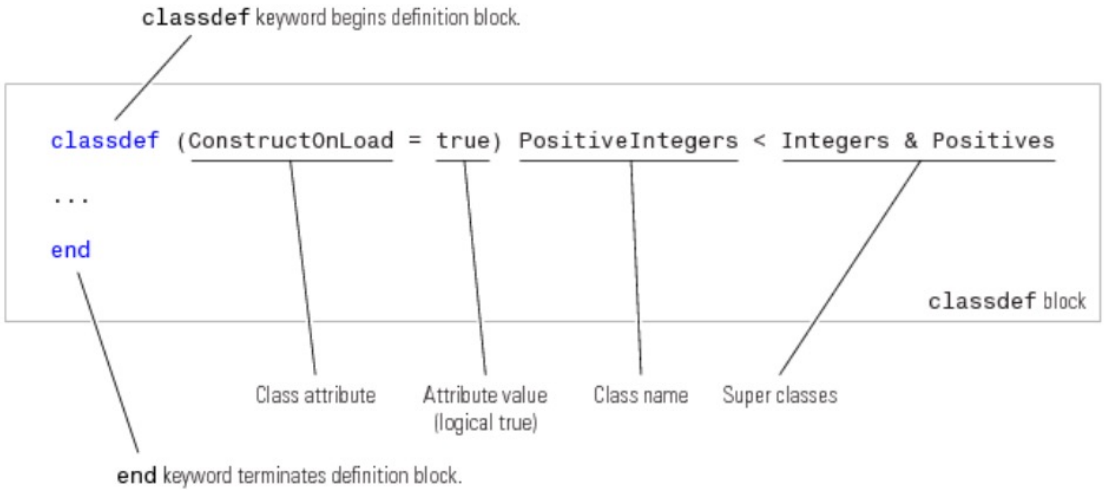
- **Class**: template for creating objects, defining properties and methods, as well as default values/behavior
- **Object**: instance of a class that has a state (properties) and behavior (methods)
- **Properties**: data associated with an object
- **Methods**: functions (behavior) defined in a class and associated with an object
- **Attributes**: modify behavior of classes and class components
- **Inheritance**: object or class (subclass) derived from another object or class (superclass)
- **Polymorphism**: single interface to entities of different types (e.g. a shape superclass - triangle, rectangle, circle, etc. subclasses)
- Other OOP features, e.g. events and listeners

Class Components

- `classdef` block
- `properties` block
- `methods` block
- `event` block
- `enumeration` block

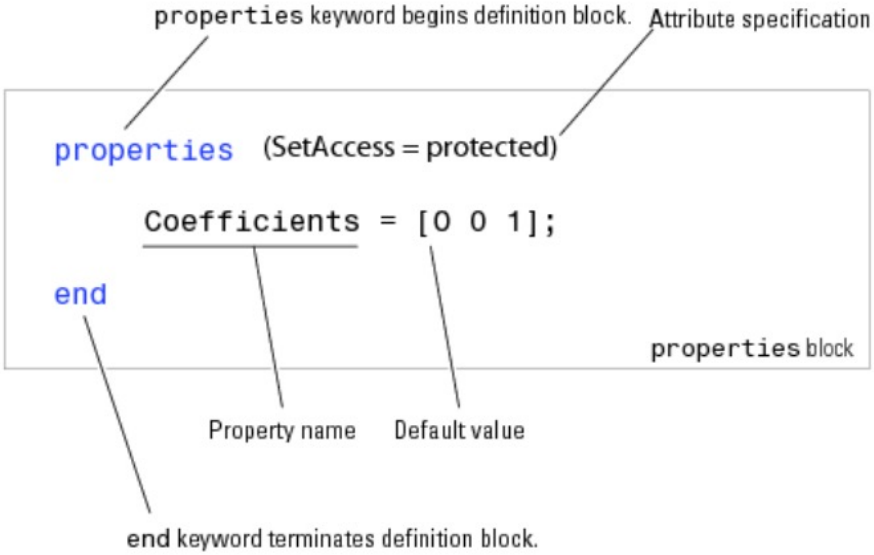
`classdef` block contains class definition, class attributes, and defines superclasses

- Specify attributes and superclasses
- Contains properties, methods, events subblocks
- One class definition per file
- Only comments and blanks can precede `classdef`



`properties` block defines all properties to be associated with a class instance; It defines attributes of all properties and default values.

- Properties are variables associated a particular class
- Can have multiple `properties` blocks, each with own attributes



class_name.m

```
classdef class_name
   properties(SetAccess=private, GetAccess=public)
      PropertyName = 'some name'
   end
end
```

```
obj = class_name
obj.PropertyName
obj.PropertyName = 'new name'
```

`methods block` defines methods associated with the class and their attributes; First method must have the same name as the class, called the **constructor.**

- Methods are MATLAB functions associated with a particular class
- Can have multiple `methods` blocks

To create an instance of a class for a list of arguments, call its ***constructor***.

- By definition, the constructor is the first method in the first method block.
- It is required to have the same name as the class
- Responsible for setting properties of class based on input arguments
- Properties not set will be given default value
- Default value either `[]` or defined in `properties` block
- Returns instance of class

```
methods
  function obj = ClassName(arg1,arg2,...)
  end
  function some_method(obj, arg1, ...)
  end
end

methods (Static = true)
  function static_method(arg1,...)
  end
end
```

Demo: a polynomial class

A value class for handling polynomials of the form

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_m x^m$$

`polynomial.m`

```
classdef polynomial
%POLYNOMIAL
   properties (GetAccess=public,SetAccess=private)
    coeffs=0;
    order =0;
   end
methods
    function self = polynomial(arg)
    function [tf] = iszero(poly)
    function [y] = evaluate(poly,x)
    function [apoly] = plus(poly1,poly2)
    ...
```

Accessing Properties in Object

Properties are accessed using the `.` operator, similar to accessing fields in a structure.

public vs. private properties

- `GetAccess`, `SetAccess` define where the properties can be queried or set, respectively
- `public` properties have unrestricted access
- `protected` properties can only be accessed from within class or subclass
- `private` properties can only be accessed from within class

Types of Methods

- Ordinary methods
  functions that act on one or more objects (plus additional data) and return a new object or some computed value

- Constructor methods
  special function that creates the objects of a class

- Destructor methods
  function called when instance of class is deleted

- Statics methods
  functions associated with a class that do not necessarily act on class objects

All methods must accept the *class instance* as their first argument.

Methods can be accessed in two ways:

- Using the `.` operator with the class instance: Implicitly passes the class instance as the first argument
- Directly passing the class instance as the first argument

Implementing Operators

- Operators such as `+`, `-`, `*`, `.*`, `==`, `<`, `>`, etc. can be overloaded for a given class

- Simply implement a method with an appropriate name and number of arguments.

- When operator such as + called, it uses the data type to determine when function is called.

Let‘s implement the following methods for the polynomial class:

- `plus` to overload the + operator to return $p_3(x) = p_1(x) + p_2(x)$
- `minus` to overload the - operator to return $p_4(x) = p_1(x) - p_2(x)$
- `differentiate` to return $p'(x)$
- `integrate` to return $\int p(x)$

Value vs. Handle Class

These are two fundamentally different types of classes in MATLAB.

An instance of a **value** class behaves similar to most MATLAB objects

- A variable containing an instance of a value class owns the data associated to it
- Assigning object to new variable copies the variable.

Conversely, an instance of a **handle** class behaves similar to MATLAB graphics handles

- A variable containing an instance of a handle class is a **reference** to the associated data and methods
- Assigning object to a new variables makes a new reference to same object

Creation

By default, MATLAB classes are value classes.

```
classdef MyValueClass
  ...
end
```

To create a handle class, derive the class from the abstract handle class.

```
classdef MyHandleClass < handle
  ...
end
```

Handle Class Methods

When you derive a class from the handle class, the subclass inherits methods that enable you to work more effectively with handle objects.

Relational methods

For each pair of input arrays, return a logical array of the same size.

- `TF = eq(H1,H2)`
- `TF = ne(H1,H2)`
- `TF = lt(H1,H2)`
- `TF = le(H1,H2)`
- `TF = gt(H1,H2)`
- `TF = ge(H1,H2)`

Event and listener methods

- `addlistener`
- `notify`

Test handle validity

- `isvalid`

Other methods

- `findobj`
- `findprop`
- `delete`

When MATLAB Destroys Objects

MATLAB destroys objects in the workspace of a function when:

- The function reassigns an object variable to a new value.
- Does not use an object variable for the remainder of a function
- Function execution ends

When MATLAB destroys an object, it also destroys values stored in the properties of the object. It frees computer memory associated with the object.

No need to free memory in **handle classes**.

- However, there can be other operations to perform when destroying an object, e.g., closing a file that the object constructor opened.

→ Define a delete method in the handle subclass for these purposes.

Basic Differences

A **value** class constructor returns an object that is associated with the variable to which it is assigned.

- Reassigning this variable creates an independent copy of the original object.
- If you pass this variable to a function to modify it, the function must return the modified object as an output argument.

A **handle** class constructor returns a handle object that is a reference to the object created.

- You can assign the handle object to multiple variables or pass it to functions without causing MATLAB to make a copy of the original object.
- A function that modifies a handle object passed as an input argument does not need to return the object.
- The copy of a handle object refers to the same object as the original handle. If you change a property value on the original object, the copied handle references the same change.

Handle class demo: `element.m`

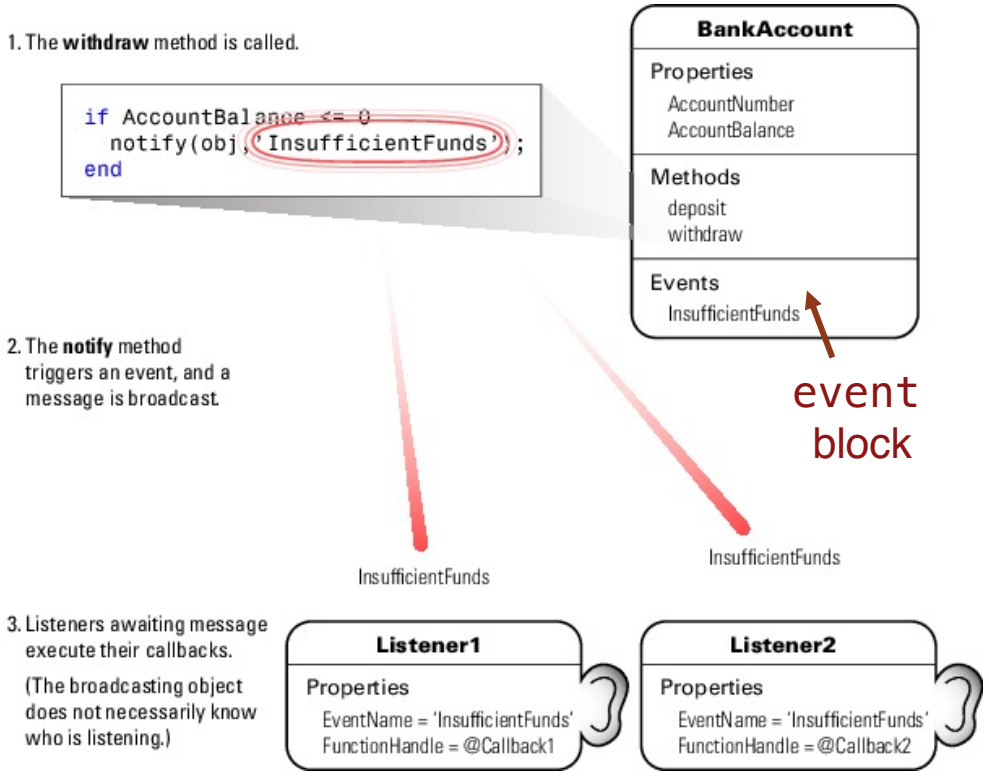
Event and Listener

The Event Model

Events represent changes or actions that occur within objects. E.g.,

- Modification of class data
- Execution of a method
- Querying or setting a property value
- Destruction of an object

Any activity that you can detect programmatically can generate an event and communicate information to other objects.



Events provide information to listener callbacks by passing an event data argument to the callback function.

- By default, MATLAB passes an event.EventData object to the listener callback. This object has two properties:
  - EventName — The event name as defined in the class event block
  - Source — The object that is the source of the event

Events can only be defined in handle classes.

- A value class is visible only in a single MATLAB workspace so no callback or listener can have access to the object that triggered the event.

Listeners

Listeners encapsulate the response to an event. Listener objects belong to the event.listener class, a **handle class** that defines the following:

- Source — Handle or array of handles of the object that generated the event
- EventName — Name of the event
- Callback — Function to execute when an enabled listener receives event notification
- Enabled — Callback function executes only when Enabled is true.
- Recursive — Allow listener to trigger the same event that caused execution of the callback.

Implementation of events and listeners requires:

1. Specification of the name of an event in a handle class — Name Events.
2. A function or method to trigger the event when the action occurs — Trigger Events.
3. Listener objects to execute callback functions in response to the triggered event — Listen to Events.
4. Default or custom event data that the event passes to the callback functions — Define Event-Specific Data.

Efficient Code Writing

Advanced Programming Techniques

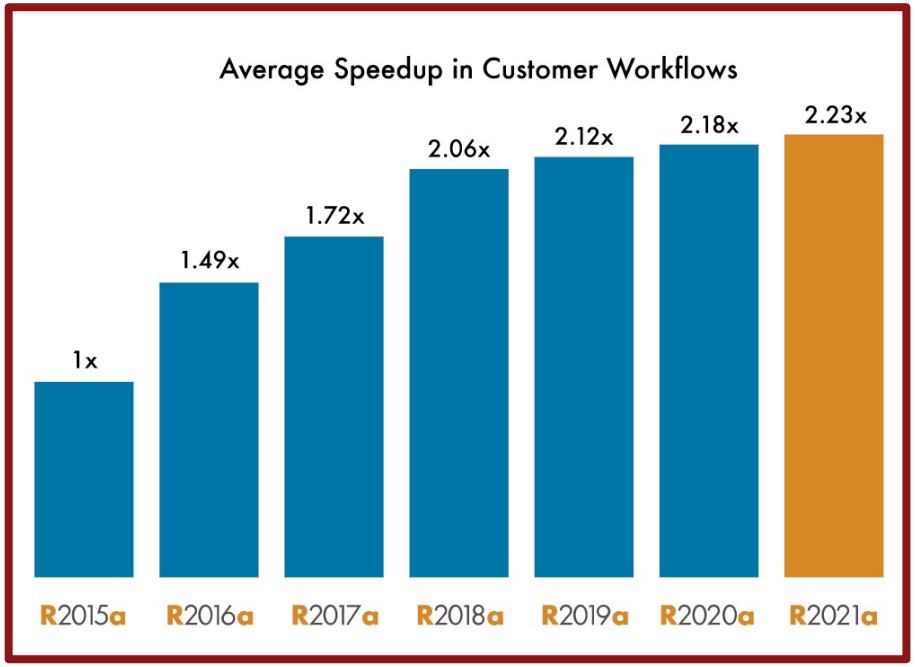


Image from MathWorks

Code Debugger and Analyzer

Syntax error

MATLAB Code Analyzer identifies syntax errors in a script or function *before* executing code.

Red

- File contains syntax errors or other significant issues.
- There is a potential for unexpected results or poor code performance.

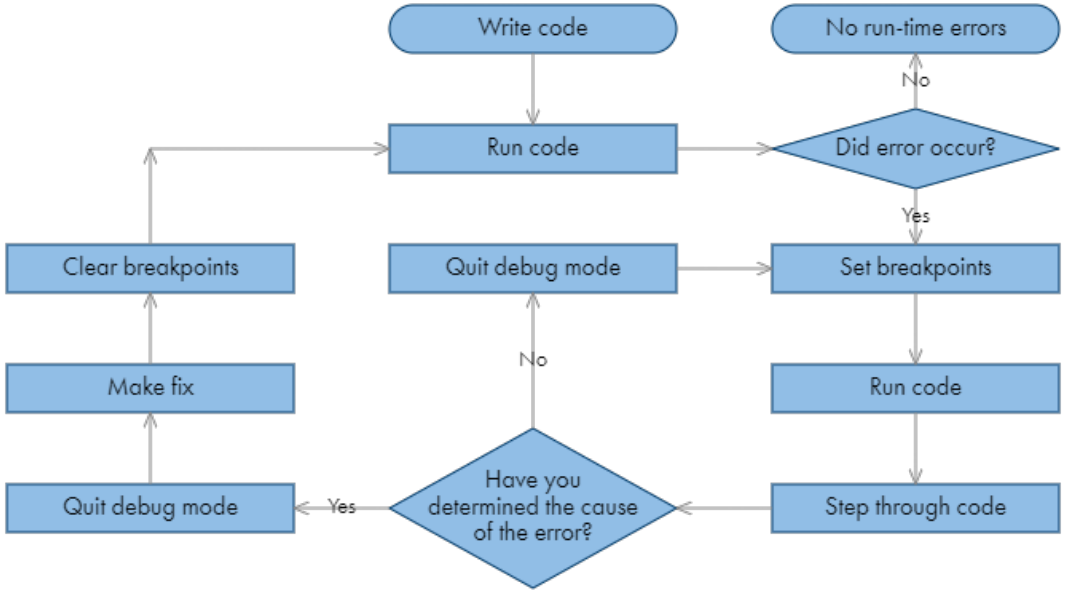
Orange:

- File contains warnings or opportunities for improvement, but no errors.
- suppress the warnings: `warning('off','all')`

Runtime error

Workflow of debugging

- Breakpoint: pause the execution of the program at runtime
- Step, Step In, Step Out, Continue



Error handling using `try/catch`

```
try
   statements
catch exception
   statements
end
```

Profiler

Debug and optimize MATLAB code by tracking execution time

- Itemized timing of individual functions
- Itemized timing of individual lines within each function

Record information about execution time, number of function calls, function dependencies.

- *The profiler report does not provide information on each individual call to a function, but rather information on the conglomeration of all calls to a function.*

→ A useful debugging tool to

- understand unfamiliar file
- find *bottlenecks*

`profile (on, off, viewer, clear, –timer)`

`profsave`: save profile report to HTML format

Other performance assessment functions:

- `tic, toc, timeit, bench, cputime`
- `memory`

`tic` and `toc`:

- `tic` starts the stopwatch and and `toc` stops it. They are affected by background processes running on the computer.
- Can run multiple times and average the results.

Quiz

Consider two functions:

```
function y = trimultiply_1(A,b) y = A*A*A*b; end
function y = trimultiply_2(A,b) y = A*(A*(A*b)); end
```

Set up the profiler to see if there is any significant difference in time performance from these two functions.
Use `A = rand(1000)` and `b = rand(1000,1)`.

Solution

```
profile on
A = rand(1000);
b = rand(1000,1);
y1 = trimultiply_1(A,b);
y2 = trimultiply_2(A,b);
profile viewer;
function y = trimultiply_1(A,b)
y = A*A*A*b;
end
function y = trimultiply_2(A,b)
y = A*(A*(A*b));
end
```

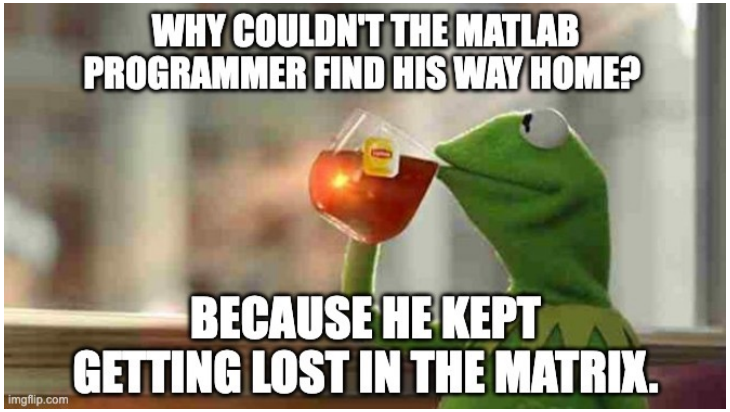
Benchmark

`bench`

measure the execution time of six different benchmarking tasks on your computer and compares the results to several benchmark computers

Task	Description	Performance Factors
LU	Perform `lu` of a full matrix	Floating-point, regular memory access
FFT	Perform `fft` of a full vector	Floating-point, irregular memory access
ODE	Solve van der Pol equation with `ode45`	Data structures and MATLAB function files
Sparse	Solve a symmetric sparse linear system	Mixed integer and floating-point
2-D	Plot Lissajous curves	2-D line drawing graphics
3-D	Display colormapped `peaks` with clipping and transforms	3-D animated OpenGL graphics

Code Performance



Optimize the algorithm itself.

Be careful with matrices!

- sparse vs. full
- parentheses: `A*B*C*v` vs. `A*(B*(C*v))`

Ordering

- Fortran ordering or arrays
- Operators with equal precedence evaluated left to right

Vectorization

- MATLAB highly optimized for array operations
- Whenever possible, loops should be re-written using arrays

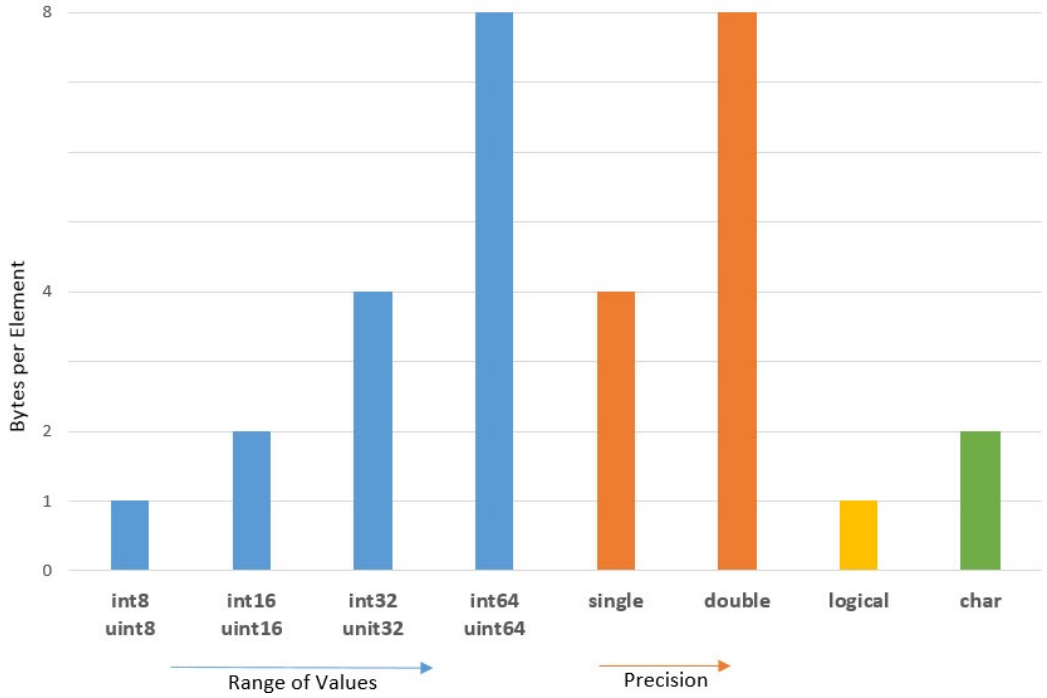
Implicit Expansion

- MATLAB will expand an array along any *dimension of length 1* to make it match the size of the other array.

→ Be cautious about unexpected expansion!

Memory management

Data Types with Fixed Memory Requirement



Container Data Types

Container variables (tables, cell arrays, and structure arrays) require overhead in addition to the data they store.

- This *additional memory* is used to store information about the contents of the variable.
- The amount of the overhead depends on the size of the variable.
  - Each cell of a cell array requires memory overhead.
  - A table requires minimal overhead for each variable and for storing table properties.
  - A structure requires 64 bytes of overhead for each field name and 112 bytes for each container. If a structure array has more than one element, each element will require additional overhead.
    So, the total memory used is *112*(number of elements)*(number of fields) + 64*(number of fields) + data*

1. Preallocation of arrays

- Data structures stored as contiguous blocks of data should be preallocated instead of incrementally grown (i.e., in a loop)
- Each size increment of such a data type requires:
  - location of new contiguous block of memory able to store new object
  - copying original object to new memory location
  - writing new data to new memory location

2. Delayed copy

- When a MATLAB array is passed to a function, it is only copied to local workspace when it is modified; i.e. *copy-on-write,* or *lazy-copying*
- Otherwise, entries are accessed based on original location in memory (see *lec6_supp.m*)

3. Contiguous memory

- Indexing column-wise is much faster than indexing row-wise. MATLAB uses **column-major** memory storage of arrays;
- Numeric arrays are always stored in a contiguous block of memory;
- Cell arrays and structure arrays are not necessarily stored contiguously. The contents of a given cell or structure are stored contiguously.

Memory Management Functions

- `clear`: remove items from workspace
- `pack`: consolidate workspace memory
  - It frees up needed space by reorganizing information so that it only uses the minimum memory required. All variables from the base and global workspaces are preserved. Any persistent variables that are defined at the time are set to their default value (the empty matrix, []).
  - Useful if you have a large numeric array that you know you have enough memory to store, but can't find enough contiguous memory
  - Not useful if your array is too large to fit in memory
- `save`: save workspace variables to file
- `load`: load variables from file into workspace
- `inmem`: Names of funcs, MEX-files, classes in memory
- `memory`: display memory information
- `whos`: list variables in workspace, sizes and types

Parallel Processing

Parallel Computing Toolbox

- Perform parallel computations on multicore computers, GPUs, and clusters.
- Desktop: install the toolbox
- Online: need access to a Cloud Center cluster

The toolbox allows you to

- Accelerate your code using interactive parallel computing tools, such as `parfor` and `parfeval`
- Scale up the computation using interactive Big Data processing tools, such as `distributed`, `tall`, `datastore`, and `mapreduce`
- Use `gpuArray` to speed up your calculation on the GPU of your computer
- Use `batch` to offload calculation to computer clusters or cloud computing facilities

The main reasons to consider parallel computing

- Save time by distributing tasks and executing these simultaneously
- Solve big data problems by distributing data
- Take advantage of your desktop computer resources and scale up to clusters and cloud computing

Some useful Parallel Computing concepts

- *Node:* standalone computer, containing one or more CPUs / GPUs. Nodes are networked to form a cluster or supercomputer
- *Thread:* smallest set of instructions that can be managed independently by a scheduler. On a GPU, multiprocessor or multicore system, multiple threads can be executed simultaneously (multi-threading)
- *Batch*: off-load execution of a functional script to run in the background
- *Scalability*: increase in parallel speedup with the addition of more resources

Installation of toolbox

- Select the checkbox during the initial installation of MATLAB
- Add-Ons → Get Add-Ons → search for the MathWorks toolbox → install

* To view the list of all the toolboxes installed, use

```
ver
```

* To check the number of cores:

```
feature('numcores')
```

Next Lecture

Image Processing Toolbox

Computer Vision Toolbox

Image Acquisition Toolbox

Signal Processing Toolbox

Audio Toolbox & DSP System Toolbox

Fun with MATLAB

Type `xpquad` in the command window…

and play around with it using the toggle bars!