71	71	71	71	71
55	55	55	55	55
99	99	99	99	99
11	11	11	11	11

1 column → 5 copies.
Column wise.

A(A < TRIC_column - 2) = 0

∴ Any element greater than its row's -2nd column element survives

0	71	0	95	86
0	55	0	0	80
0	99	0	0	0
35	11	0	95	92

→ Avoids lots of explicit loops!

---

* Locate odd elements in a vector of integers

mod (90, 2) → 0

v = [81 90 13 91 68 10 28 55 95 96]

mod (v, 2)

ans = [1 0 1 0 1 0 0 1 1 0] → Not logical (numerical)

~(ans) == 0 (even)

1x10 logical array

1 0 1 1 1 0 0 1 1 0

find(ans)

1 3 4 5 8 9 → index.

a(ans)

81 13 91 63 55 95

Note: array indexing - done only using +ve integers
(or)
logical index! (logical array)

```
[row, col] = find (mod (A, 2) == 1);
```

```
>>[row, col]
```

1	1
3	1
1	2
2	2
2	3
3	3
3	4

→ gives all elements row, column number (satisfying)

81	91	28	96
90	63	55	16
13	10	95	97

---

other speed up

* Recursion: Stop! (∵ elegant but performance penalty! - most of the time iterative problem: easy).
* visiting all folders
  sub folder tree
  Sorting lists (Binary search

) easy than iterative one!

→ 'speed up'!

---

modes of passing arguments

* call by value: copy (stack) - function access as local variable.
* time & memory to copy!

Trick: Argument passing as reference!

* matlab watches keenly during execution: every statements to change - even 1 element out of million - make a copy in stack & continue execution'. → call by reference abandoned
  Switched to call by value.

* Avoid changing - Large arrays unnecessarily

function mx = input_mod_test (A)

mx = max(A(:));

end

x = randn (1e6, 2e4);

→ timeit (@() input_mod_test (x))

ans = 0.1537

---

mx = max(A(:));

A(1) = 0;

→ Now ans = 0.1470

↳ As element changed: executed call by value (∴ more time).

Binary search → ≈ c log(N)

only when:

'Call by reference' → 'Doesn't change its values'

Call by value → N copy operations → O(N)

Index reordering

ictan pre allocation - not possible

for ii

for jj

A(jj, ii) = ...

→ change

for jj = 1:M

for ii = 1:M

A(ii, jj) = ...

(or)

for ii = 1:M

for jj = 1:M

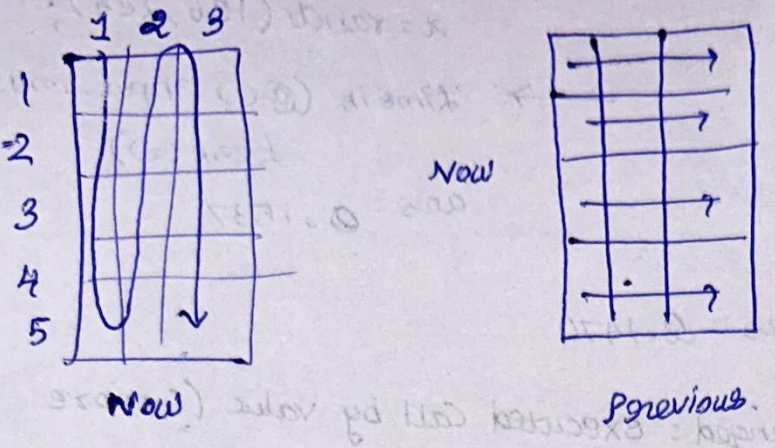
A(ii, jj) = ...

why: Now → MATLAB knows range

Not

(Allocates every time) → Saves time!

before: 21.906533 → 3000
After: 0.282243 → 3000
) why! (Not everytime the array is reallocated.



Less no. of reallocation'

Row major → But not time saving!

Column major → less reallocations!

Row major – time(more) – 3.2578
column major - low time – 0.9946

---

why! Fortran: Column major

Fortran follows column major (1954: First high level language for numerical Computing).

↳ designed: map array elements order in the memory in column major order.

---

* Why MATLAB: People moving from FORTRAN to MATLAB wanted that!

* C, C++, → uses row major order!

parfor : parallel for loop — Introduced in 2012

Available — parallel Computing Toolbox

only when multiple Cores?

Run iterations in parallel. (Parallely)

---

* Caution: Body of the loop can't depend on earlier iterations of the same loop.

* parfor only works if the body can be executed in any order with no effect on the outcome.

$$V(ii) = V(ii-1) \longrightarrow \text{Not possible.}$$

Other problem

Parfor makes only sense - If the body of the loop is slow enough to make up the extra time generated by parfor for the overhead of dividing up the work across multiple Cores,

In scientific Computing

Body of a for loop is often very slow because it does a large no. of complicated computations. (repeatedly on random data / independent data).

Calculate Eigen vectors

```
function a = eigen_for(A3D)
    a = zeros(1, size(A3D, 1));
    for ii = 1:length(a)
        a(ii) = max(abs(eig(squeeze(A3D(ii,:,:)))));
    end
```

↓

```
>> A3D = rand(504, 50, 50);
>> tic; af = eigen_for(A3D); toc
```

↓

25.932406 seconds.

Before: → (parallel computing toolbox) installed!

↳ Choose start parallel pool.

tic; ap = eigen_parfor(A3D); toc

8.771132 seconds. Each core: 1/4 of iterations (factor of 4 speed up)

↓ may be low (preparation)

OOP - when large projects.

* Larger size - procedural programming - more & more difficult.
* million of data - having global & local variables, functions - harder to keep track of the data. (modify etc..)
* OOP - Solution.
  * "OOP - centered in data not function"
  * "Data - stored inside an object"
  * object - struct with extra features.

eg:

```
>> VeryCoolGuy = struct('name', [])

Struct with fields:

    name: []

>> VeryCoolGuy.name = 'Mike'

    name: 'Mike'

>> VeryCoolGuy.game = 'professor'

    name: 'Mike'

    game: 'professor'
```

'Why Object is like a Struct?'

* Object also includes one/more fields.
* data of an object - stored in those fields.

'.' operator.

Additional features:

* Object can contain function.
* Object also stored in a variable
* object - defined by a user (type) → user defined type - classes known as

classes are defined using keyword 'classdef'

```
classdef contact
    properties
        FirstName
        LastName
        PhoneNumber
    end
end
```

→ classdef is legal only if it is saved in a .m file with same name as the class.

---

person = Contact → Not assigns just a copy of object but creates a new object of Contact type.

'creates an object of Class Contact'

It's not possible to assign a Class to a variable - only able to assign an object of the class. (creates an object and assigns to variable)

Assigned values default to properties = [ ]

* 'Every property of every object must have a value at all times'

*

>> person.FirstName = 'Mike'

>> person.LastName = 'pat'

>> person.PhoneNumber = "(555) 123-4567".

'Not able to assign like this' – lot of variables (objects)

* Command – Introduce features (function!)

classdef

properties
methods ← Simula (1960s) → methods (introduced in this language)

In C++, Java (today) – class function – known as – member function.

---

classdef

properties
methods

Constructor ⟶ a method that creates an object & gives values to its property at the same time.

o/p of Constructor – New object it creates!

* 'No restriction – in no. of Arguments' / can use varargin
* like an ordinary function.
* variable number of i/p arguments important for MATLAB.

> In MATLAB ⟶ only one constructor is allowed in a class

>> * MATLAB - A single constructor - allowed in - class.

* we can't use property name by itself! (struct) - same here

obj. First Name | Can't | First Name!

*

>> person2 = Contact ('Lede', 'AKOS', 555972);

Added advantage - we can control type too

```
function obj = contact (lname, fname, phone)
    obj.Last Name = String (lname);
    obj.First Name = String (fname);
    obj.Phone Number = String (phone);
end
```

"made sure - everything is string"

---

what's the big deal!

person2.LastName = pi;

First Name = "AKOS"

Last Name = 3.1416

Phone Number = "5523476"

] can break other parts of program. Since assumed as String".

Ten's of thousands of class → Hazard!

*Design a class - Users can't violate!

* preventing somebody! - From messing!

Access!

Whenever we access the properties - it will call the access method & check!

```
function obj = set.LastName(obj, lname)
    obj.LastName = string(lName);
end
```

→ Property set method!

```
>> person2.set.LastName('Smith') ⟶ error
```

we can't directly access set methods!

```
>> person2.LastName = "Smith"

person2 =

    FirstName = 'Akos'
    LastName = 'Smith'
    phoneNumber = '55992'
```

matlab automatically checks for any set methods & implements it. (unlike Java).

```
>> Person.LastName = pi

LastName = "3.14..."
```

must!

```
function obj = setLastName(obj, lname)
```

↓ dot operator

eg:

Set_LastName → Won't work.

get methods

```
get.LastName(obj)
```

* Set access methods frequently used
* get access methods – don't !

'Store more' !

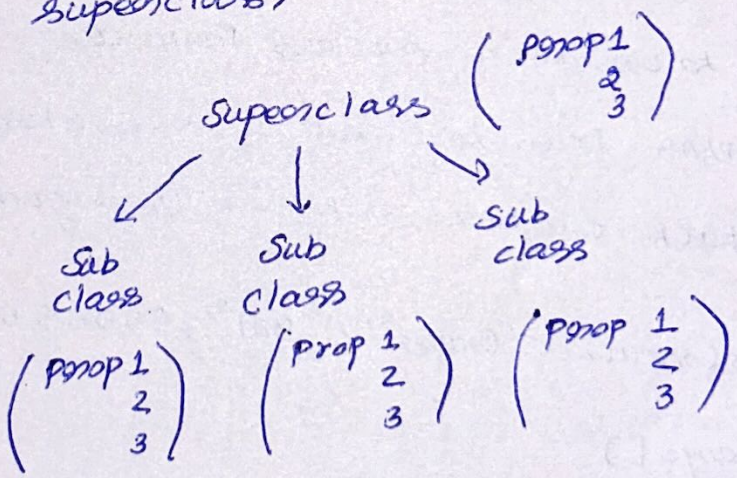
* Business Contact – Need Company name
* we can Company name – not used for personal Contacts (empty always)
* create new class – by copying Contact class + new features !

'Not a great idea'

New business class – that has properties of the existing properties of Contact class – relevant to its purposes – without any copying.

Important feature of OOP

* one class can be built on other class. (Inheritance)
* create a class – specify – a new modified / derived version of other class – such class is known as subclass. (Inherits all the properties of superclass)



* Superclass (Base class)
* Subclass – modify methods that inherits & add its own class.

How?

```
classdef BusinessContact < Contact
    properties
        Company
        Fax
    end
end
```

>> b = BusinessContact ⟶ error! → Not enough i/p arguments

Error in Contact

obj.LastName = string(name);

Remember: how do we get construction of the Contact class - when we were constructing a BusinessContact object.

Note: while creating subclass - instance of superclass also created!

Here: Trying to create a subclass instance.

* MATLAB: Tries to create a superclass instance first.
* Contact construction has 3 i/p arguments.

b = BusinessContact("Gates", "Bill", 555555);

Company: []
Fax: []
FirstName: "Bill"
:

Create a set method

make → set method — get any no. of i/p arguments!

using nargin

if nargin < 3 phone = "" ; end

⋮

"So no problem – even without called with enough i/p arguments"

Contact class

"not set to work with zero arguments"

classdef BusinessContact < contact

⋮

methods

function obj = Bus..Con.(5 i/ps)

obj.LastName = String(lname);

⋮

end

end

end

---

* won't work! → MATLAB calls superclass – but superclass must need three i/ps.

* So we can't able to create instance of superclass in subclass "error"

Now:

use nargin,

configured to accept empty arguments (zero)

Now

Company = "MS"

Fax = "555 98"

⋮

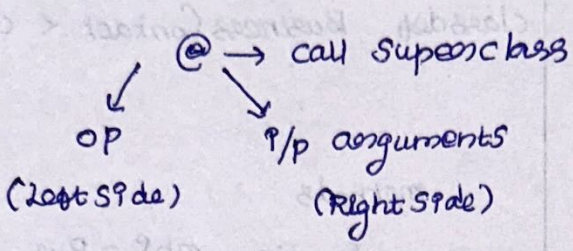
Here: MATLAB calls with zero i/p argument

creates [] superclass

In BusinessContact function → we updated them with values!

Call superclass by ourselves (Instead by default).

```
obj @Contact (lname, fname, phone);
obj.Company = string(cname);
obj.fax = string(f);
```



Note: Superclass must be called before using subclass variables

---

Handle class –

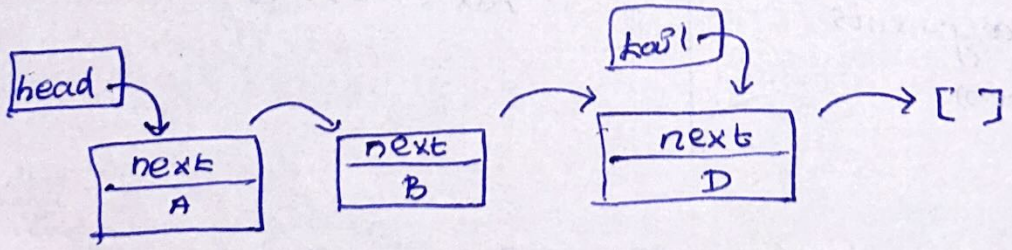
* MATLAB – built around matrices
* Disadvantage – Fixed size (one contiguous single memory) – often unnecessary
* Reallocation – more time & space! (Not enough room for new) – Behind the scenes.

Slow our program way down!

Sometimes: Not possible

Solution: Another data structure

* Linked list.



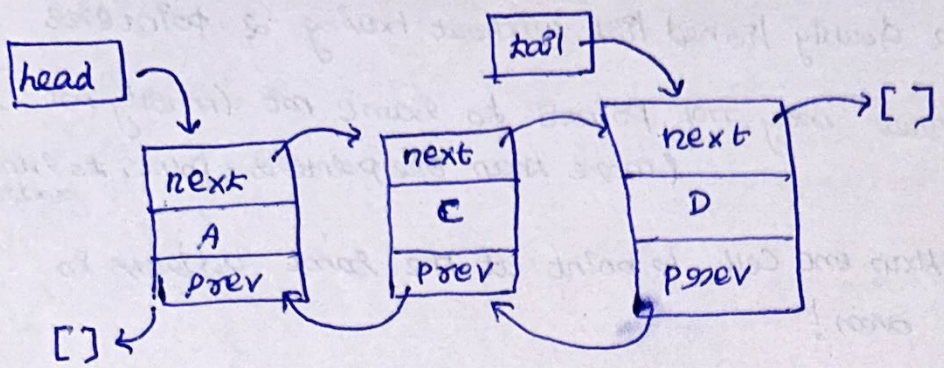
Set of individual elements linked into a chain.

Each element (in Computer Science terminology – node)

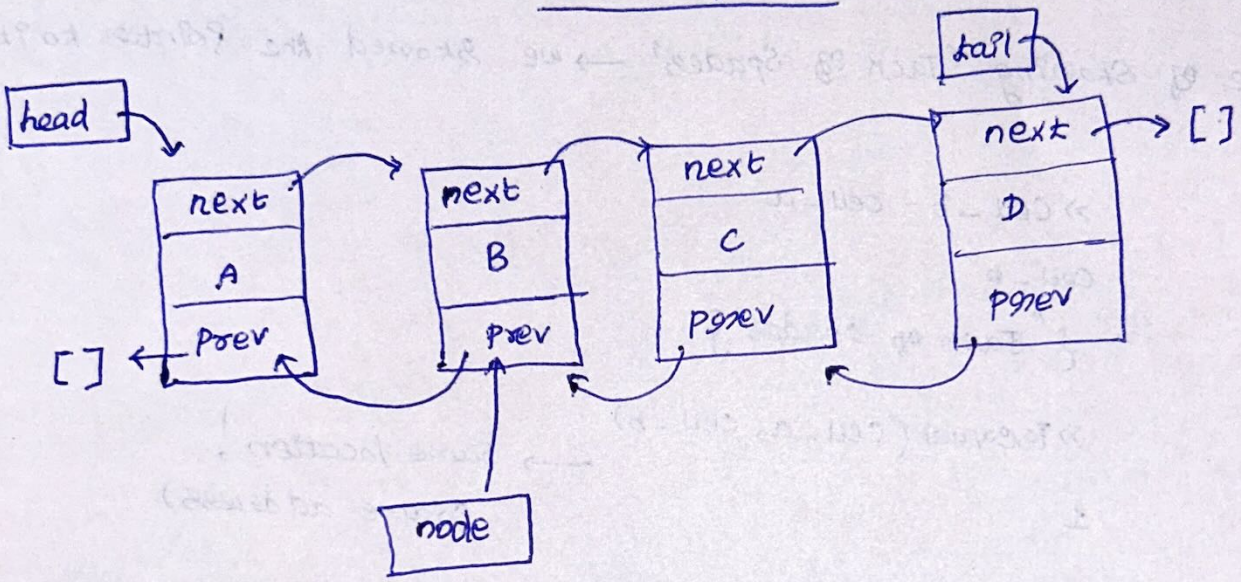
* next → points to next element.
* Zero (when no upcoming nodes) → Matlab [ ]

From head → move until next == [ ] (tail)

Doubly linked list



Insert (update)



* We can either remove node / leave as it is — depending upon the need!

---

* Beginning / end → we need to re-enter head/tail

Problem!

* datatype eg head, tail, next, prev → other languages (pointers)
* MATLAB – cell (But not close enough) – Failure
* Note: A's next pointer, C's prev pointer → points to same node (B)

* In diagram ⇄ → Just doesn't representation.

* only one node! — only one address!

Point: can't implement: a doubly linked list without having 2 pointers

∴ case: next & prev may not points to same not (mostly not!)

(more than one pointers - points to same address)

---

* MATLAB: won't allow more than one cell to point at the same address in memory. with cells: Game over!

* cell_a = {'Jack of Spades'}

* Inspite of storing 'Jack of Spades' → we stored the pointer to it

```
>> cell_b = cell_a
cell_b
{'Jack of spades'}.
>> isequal(cell_a, cell_b)
1
```

→ Same location! (Same address)

```
>> cell_b{1}(1:4) = 'King'
>> cell_b
{'King of spades'}
>> cell_a{1}
'Jack of spades'
```

even though cell_a & cell_b points at the same thing → it's not!

* Instead of copying Cell-A into Cell_B → matlab made of Copy of the String- 'Jack of spades', & then returned the Copy to the Cell-B.

So Values of Cell-A & Cell-B → same

pointers → Not

Moral: No two variables → points to same memory location

* Equal → means same data objects they are pointing!

*

"No two cells points at the same object"

How to Implement?

Handle class

* Implement double linked list!
* function handles – allows an access to a function!
* file handling – access to file!

handle-class – access to an object!

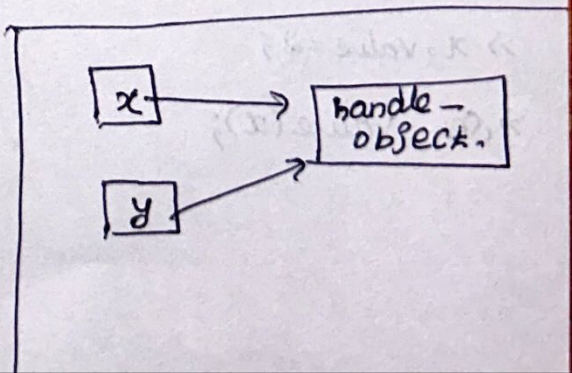
assign a handle object to a variable:

* It assigns the reference (pointer) to the object instead of variable itself. (Just the way a cell variable gets a pointer). But with handle object – no need for those special {}
* The pointer that it automatically gets is called a handle. Handle includes the address of the object along with some other info.

```
x = handle_object;
↓
handle variable. (Similar to cell variable)
```

Difference: Allows two handle variables to point at the same handle object.

```
x = handle_object;
y = x;
```



Handle class

```
classdef AKos < handle
    properties
        Coord = 'Jack of Spades';
    end
end

OtherGuy = AKos
OtherCopy = OtherGuy
OtherGuy.Coord(1:4) = 'King'
OtherCopy.Coord
```

'King of Spades'

changed.

non-handle class

```
classdef Mike
    properties
        Coord = 'Jack of Spades'
    end
end

CoolGuy = Mike
CoolCopy = CoolGuy
CoolGuy.Coord(1:4) = 'King'
CoolCopy.Coord
```

'Jack of Spades'

'Not changed'

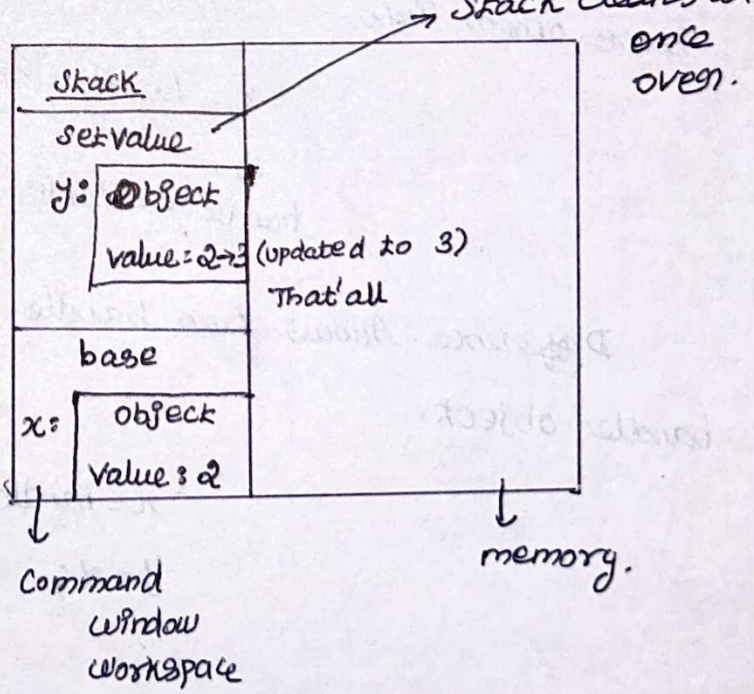
Keyword: classdef (func. name) < handle

Consequences:

SetValue.m

```
function Set_Value(y)
    y.value = 3;
end
```

```
>> x = valueclass;
>> x.value = 2;
>> Set_Value(x);
```



'The change-doesn't reflect: Since its pass by value'

Same experiment with handle variables

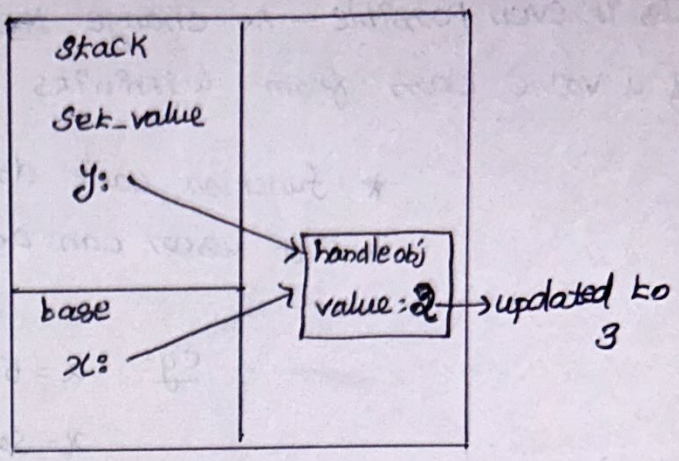
```
>> x = Handleclass;
>> x.value = 2;
```

ex gets only pointer

```
>> set_value(x)
```

(passed the reference)

```
>> x.value
3.
```



```
classdef Testclass
    :
    methods
    :
        function set_value(obj, val)
            obj.value = val;
        end
    :
end
```

set.value (actual)

Now!

brandon.set_value(3) } since pass by value.

2 → not updated

```
>> brandon.set_value(3)
>> brandon.value
    2.
```

why: since object not given!

obj.value = 3 (created)

But not returned!

So (2)

```
>> brandon.set_value(obj, 3)
2
```

why: pass by value, not returned

obj copy is given as argument

we need handle class - else return!

Is it even possible - to change the value of the property of an instance of a value class from within its own methods!

* function can't do it!
* But user can do it!

eg: x = 5;

x = sqrt(x)

2.2361

Now

```
function obj = set_value(obj, val)
    obj.value = 0;
end
```

↓

Now

brandon.set_value(3) ⟶ ∴ returning!

else

set.value(obj, val).

in function

3

without handle

```
classdef Testclass
    properties
        value
    end
    methods
        function obj = Testclass(val)
            if nargin == 1
                obj.value = val;
            else if nargin == 0
                obj.value = 0;
            end
        function set_value(obj, val)
            obj.value = val;
        end
    end
end
```

with handle

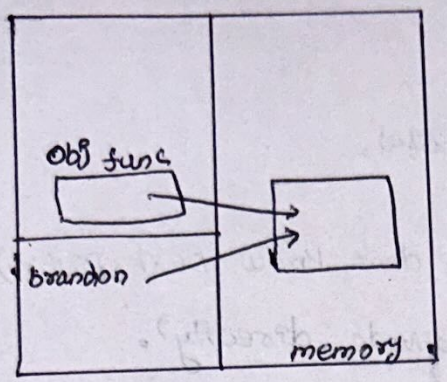
```
classdef Testclass < handle
    "
    "
    "
    "
end
```

>> brandon.set_value(3) → why ∴ function

3

brandon = TestClass(2)

(pointers)



'Same handle' → so even though object not passed, reflected in answers.

'Super powered'

Now: handle class → can change values in workspace → Be careful. (even without knowledge!)

'use . operators' → for differentiating (precaution).

★ Implementation eg double linked lists

* order eg next, prev changed
* names can be capitalized
* other properties can be added!

★ list: multiple nodes + heads & tail

2 classes

* nodeclass (unit)
* doublylinkedlist class (whole)

Superclass – LinkeNode

LinkedNode.m

```
classdef LinkedNode < handle
    properties
        prev
        Next
        Owner
end
```

methods

```
    function node = LinkedNode()
        node.prev = [];
        node.Next = [];
        node.owner = [];
    end
end
end
```

* Owners → provide access to linked list that has the node

Why?

each node – only one next
only one handle.

* A node can be in only one single linked list.
* Having a handle – to access list will be useful.

---

* we don't need to reset (At starting, we don't know next, prev)

* we never want to create an object of type LinkedNode directly?.
* we want to be able to design subclasses of it.
* Those subclasses will specify: what data they can store.
* But inherit the behaviours from LinkedNode class.

A node can't be linked into two nodes at once?

```
node.owner.remove(node);
     ⌣
Front.remove(node);
```

Left to right associativity.

---

making private

Properties (Access = private)

Head
Tail
length
end

A property.

either: public / private : not both same at the time.

No declaration: public (default).

possible to have both public (some public properties & some private properties) in the same class definition.

*privacy declaration: objects already defined objects.

"If a class is private - problem with its subclasses, objects created previously"

↳ "Not all the subclassed becomes private" → wrong!

---

* Worst case: Insert at end.
* Need : Sorted order (Better). (Increasing order)

* LinkedNode → doesn't store any data! [own example]
* Compare new item with existing items! (general way: insertion works for any subclasses - may added later)

operation overload!

operation overload - Redefine - what existing operators do when applied to instances of that class.

Sorting - redefine the relational operators (<, ==, >)

→ only to that file (Allowed access)

Properties (Access = ? Dlock)

---

gt ( ) → Matlab greater than function.

↳ called whenever > is used! (MATLAB calls in back)

If we define a method named gt in a class, the MATLAB will invoke that method instead of the built in gt function - whenever you compare two objects of that class with a > operator.

"operation overloading"

---

* "Defining a method to redefine - a built in function for a specific class of i/p arguments is called overloading."

* When the function is the implementation of operators like ">", "<" or "==" → it's known as operation overloading!

Abstract attributes

Assign abstract attributes to methods as we have here,

methods (Abstract) (or) properties / classes themselves.

```
    fx (a,b)
end
```

purpose:

* we don't want anybody to create an instance of the link node class.
* Instead we want to restrict them to creating instances only of subclasses of link node.
  - ↳ done that (enforce) - restriction by marking the link node class being abstract.

---

3 ways!

* Include in heading
* make any of its methods abstract
* make any of its properties abstract.

once a class is marked abstract - we can't create any instances of it.

```
>> A = LinkedNode
```

Abstract classes can't be initialized!

---

* Furthermore, if you define a subclass of a class with any abstract methods or properties - then subclass will also be an abstract class - with no instances allowed unless - it includes a concrete implementation of every abstract method & every abstract property in its superclass.

---

arguments block: (must) - 4 rules (see pdf)

optional: qualifiers

```
n (1, 1) {mustBeNumeric} = 0;
```

n (1, 1) — Brown — {mustBeNumeric} = 0;

- n (1, 1) → size
- {mustBeNumeric} → validation function! (see documentation)!

Including conversion type

```
arguments
    n(1,1) double {mustBeNumeric} = 0
                                  ↳ default value
end
    node.value = n;
```

eg: int8 can be converted to double!

Can't be converted / violated (any) → Red error is thrown execution halts.

---

we can add default value (if actual argument is omitted)

```
n(1,1) {mustBeNumeric} = 0
                       ↳ default
```

* If small size i/p → Expanded to mentioned size (execution continues)
* arguments → Can't be used in abstract methods / nested functions

---

gt (a,b) → greater than

ge (a,b) → >=

lt → <

le → <=

eq → ==

ne → ~=

Protection:

* Private properties → not accessible to subclasses
* Just change the access to public

middle: protected → Access to all subclasses of a class

Applications

* Without MATLAB - lets run as app! (standalone application)
* GUI - Graphical User Interface! (It should have a GUI)

App designer — Build an app

'berkeleyearth.org' — Data overview

1 → year, 2 → month, 3 - estimated global average temperature

---

2040 months (Jan 1850 to DEC 2019)

* 'made: year & month to single value' - For processing!

* '0 radio buttons' — one option at a time
* Stop year before start year — don't allow (keep the knob correct)

Average over fewer months - not so good looking (Not any notable change than org.)

120 months rounded up — stops at 2015 (5 years before & after data)

∴ So need that much data for windowing (120 months).

* Default: restores to initial position!

Open app designer

* New → app (or) Apps - Design app

middle - canvas (design GUI layout)

Title: Global Temperature Anomaly.
Xlabel: Year
Ylabel: C (relative to 1950 - 1980 mean)

mlapp → matlab app

Call back functions: Called when certain events occur in our app.

* 'do something: while mouse - inside a widget' (Function triggering)
* clicking a mouse button / hitting a button.
* start / stop app - can trigger call back

'No need for functions for call back function'

Startup Fcn: Startup function (Task after app is started)

* Select default name.
* popup - code - else click Codeview.

---

classdef GlobalTemps < matlab.apps.AppBase

↳ 'Name of folder'

'Automatically generated code'

---

* 'can't able to type in grey areas' !

* when an app starts - creates an object of the class GlobalTemps
* Construction is called immediately. (Local name of that app)
* 1st create axis - then registers app with the app designer.
* Then executes by startup function

runStartupFcn → done this job!

app function handle (two variables).

runStartupFcn Construction:

Coolguy = 'MIKE' ; → In code view

Command prompt - Run

>> GlobalTemps

Coolguy =

'MIKE'

'The Constructor has been called!'

To avoid conflicts with functions on the path, specify variables to load from file

load GlobalTemp.mat

(data)

↓

'Though - Sure I won't name function as 'data'.'

* MATLAB: Always one step ahead - precaution!

more specific

load GlobalTemp.mat data;

data → 1col (time)
→ 2col (Temperature)

Plot data

plot (app.UIAxes, data(:, 1), data(:, 2));

↳ 'property variable - UI axes that has the handle to the canvas for axes'.

Note: data - local scope - can't be accessed by other commands other than startupFcn function!

Note: we need that data over & over!

store: Somewhere else - properties

Problem: property section: grey!

Property window: (menu)

* No idea to give access - outside-app (so let it be private)

```
properties (Access = private)
    Temperatures    % global temperature (in C)
    TimePoints      % measurement times (in years)
end

function StartupFcn (app)
    load GlobalTemp.mat data;
    app.Temperatures = data(:,2);
    app.TimePoints = data(:,1);
    plot (app.UIAxes, app.TimePoints, app.Temperatures);
end.
```

Radio button

Data Source

- o Monthly
- o Monthly Average
- o Both ⟶ Inspection ⟶ Value (click)

e Tells - its default?

Knob - Stack

Limits ⟶ 1840 to 2020

Initial value ⟶ 1850

Values

MajorTicks ⟶ 1850, 1910, 1960, 2020

(data vector)

Step - Knob

default: 2020 6, 30, 60, 90, 120

Now: Nice looking facade

* Time to give life by adding - call back functions

Step:1: click radio buttons - (rightclick) → call backs.

we want: do something when user pushes button.

* SelectionChangedFcn callback

Name: DataSource ButtonGroupSelectionChanged → can't change (grey)

Name (button) — Callback functions.

→ That property holds a handle.

* Note: Conversion - int → double
  cell → (can't be) in to a double.

] Take care by MATLAB (Inbuilt properties)

---

* we can define a class inside its own class folder which has same name as the class itself but prepended with an at sign (that folder can be contained - in a package folder - can have any name as long as it starts with plus sign")

"Packages can be inside packages" → and can access the class

package.subpackage.....class (Finally)

---

* ∴ monthly button - automatically assigned that button. (object)

* put the cursor there

Value: 1 → (logical: one currently selected)

Text: 'Monthly'

position: [11 60 65 22] → coordinates in pixel.

* app. DataSourceSelectionGroup. SelectedObject; → have details

* switch app. DataSource ButtonGroup. SelectedObject;

case app. MonthlyButton

fprintf ('Monthly pushed');

Case app. Monthly Average Button

fprintf ('Monthly Average pushed \n');

otherwise

fprintf ('Both pushed \n');

end.

---

only when buttons changed = effect.

∴ name: SelectionChanged (. . .)

otherway - know whether key is pressed!

* each radio button has value - 1 if & only if it is pressed

if app. MonthlyButton. value

"

else if app. Monthly Average Button. value

"

else if app. BothButton. Value

"

Simple way

function DataSourceButtonGroupSelectionChanged (app, event)

↓

old value

our need! ← new value

Source

Plot ⟶ Both

↙ monthly data ↘ monthly average

Two properties → Show Monthly (private) → Add with Temp, Timepoints.

→ Show Average.

[Previous: done using property button → This time matured!]

★ Initialize these properties in Startup

app.ShowAverage = app.MonthlyAverageButton.Value || app.BothButton.Value;

app.ShowMonthly = app.MonthlyButton.Value || app.BothButton.Value;

edit – Instead of printing – do something

(In button – functions)

event.NewValue

Switch ~~app.Plotting~~

Case app.MonthlyButton

⋮ Plot code!

⋮

. . .

Before new plot – refresh old plot

↓

Do in DataSourceButtonGroupSelectionChanged

★ monthly: only monthly

★ Average: only average

Function – does the plotting!

'we don't need a callback' – we need a normal method – call backs can call!

'Own function: No o/p'

```
methods (Access = private)
    function plotData(app)
        plot(app.UIAxes, app.TimePoints, app.Temperatures);
    end
end
```

↓

Need

```
if app.ShowMonthly
    plot(app.UIAxes, app.... , app.Temp..);
end
if app.ShowAverage
    Compute Average!
    plot window
end.
```

```
function computeAverages(app)
    app.Averages = movmean(app.Temperatures, 30) % (Calculate moving mean)
end
```

∴

```
if app.ShowAverage
    plot(app.UIAxes, app.TimePoints, app.Averages, 'r',
        'LineWidth', 2);
end.
```

'hold-on, hold off' → else deleted.

hold(app.UIAxes, 'on');

hold(app.UIAxes, 'off');

Add calls in - Startup

app.computeAverages(); → one time enough!

app.plotData();

Everytime changed!

"Do in radio button callback"

app.plotData();

Title → edit in Identifiers.

Colors

0.35, 0.56̄, 0.56 — 16,777,216 Colors

Colors & transparency (axes)

Bg Colors → 0.78, 0.76, 0.73

↓

Do for radio buttons, etc...

Knobs — callback

Add ValueChangedFcn callback → only once (released)

Add "Changing" " → gets called repeatedly

∴ plotting - Few 10's of points → very quick

"we can afford - repeated calling"

* For button: Nothing really changed: So no variable (By MATLAB)

Store year

StartYear

StopYear

data(1:10,1)

1800

1801

⋮

1807

xlim(app.UIAxes, app.StartYear, app.StopYear);

xlim(app.UIAxes, 'manual'); → Freezes the axes.

Problem: StartingYear not initialized (In startup fun (call back)

app.

Constrain

StartYear > StopYear

need to be with in limit!

if app.StartYear > app.StopYear - 1

app.StopYear = min([app.StartYear + 1, app.StopKnob.Limits(2)]);

app.StopKnob.Value = app.StopYear

app.StartYear = app.StopYear - 1;

app.StartKnob.Value = app.StartYear;

end.

atleast 1 year

1950, 1951

my aim

1960 → 1959

1960 (Fixed)

where is the graphics? → In Class definitions eg widgets.

```
function StopKnobValueChanging(app, event)
    app.StopYear = event.Value;
    if app.StopYear < app.StartYear + 1
        app.StartYear = max([app.StopYear - 1, app.StartKnob.Limits(1)]);
        app.StartKnob.Value = app.StartYear;
        app.StopYear = app.StartYear + 1;
        app.StopKnob.Value = app.StopYear;
    end
```

1950 1950

1949 1950

Note: within limit

∴ start limit can't be greater than limit

(decreasing)

1850, 2020

max(, value)

1950 1949

1950 1951

min(2020,)

Increasing!

Case: Not same knob limits.

1850 1900 1870 1900

Stop = min(Startyear + 1, Stop limit(2))

app.AverageWindowSlider.Value = app.AverageWindowSlider.Limits(2)/2

⋮

Default ...

---

ylim(app.UIAxes, [-1.5, 1.5]);
ylim(app.UIAxes, 'manual');
→ Better (No Space - Compact!)

* moving average function — slides a window across data
* Biased — (∵ usual: equal no of points to left & right) → else biased.
* window centered around current data point.

> Beginning: No data to the left
> end : No data to the right

∴ movmean — uses data from the right → ∴ biased!

* move to half — add more & more data points to the left as they become available. → 'Need: equally weighted'.!

eliminate — slow

'Simply not plotting averages-'

* No. of points — cutout — depends upon the width of the window & how close to the center of it is in the range.

* 'Read the movmean — documentation'

```
if app.ShowMonthly
    plot(app.UIAxes, app.TimePoints, app.Temp...)
    hold(...);
end
```

```
if app.ShowAverage
    half_window = app.AveragingWindowSize;
    Start = floor(half_window) + 1;
    Stop = length(app.TimePoints) - (ceil(half_window) - 1);
    Inside_range = Start:Stop;
    PartialTimePoints = app.TimePoints(Inside_range);
    PartialAverages = app.Averages(Inside_range);
    plot(app.UIAxes, PartialTimePoints, PartialAverages, 'r', 'LineWidth', 2);
end.
```

* widgets - activepaarts - own picture (moving paarts) with call back func.

Standalone application

>> deploytool

SARS Cov 2 - Johns Hopkins University - Covid_data

2 - axes !

Google !

Cumulative

Daily — calculate from cumulative value ! (Averaging)

---

click button — change colour

* uisetcolor(app.CovidTrackerUIFigure, 'Select UI Figure Color');

app name.