

Different kinds of abstraction

- Packaging **procedures** (program **instructions**) into a **function**
 - A program is a set of functions executed in the specified order
 - Data is passed to (and from) each function
- Packaging **data** into a **structure**
 - Elevates thinking
 - Reduces the number of variables being passed to and from functions

All possible (i,j,k) combinations but avoid duplicates.

Loop index values have this relationship $i < j < k$

i j k

```
1 2 3
1 2 4
1 2 5
1 2 6
1 3 4
1 3 5
1 3 6
1 4 5
1 4 6
1 5 6
```

$i = 1$

```
2 3 4
2 3 5
2 3 6
2 4 5
2 4 6
2 5 6
```

$i = 2$

```
3 4 5
3 4 6
3 5 6
```

$i = 3$

```
4 5 6
```

$i = 4$

```
for i=1:n-2
    for j=i+1:n-1
        for k=j+1:n
            disp([i j k])
        end
    end
end
```

Still get the same result if all three loop indices end with n ?

A: Yes

B: No

i j k

1	2	3
1	2	4
1	2	5
1	2	6
1	3	4
1	3	5
1	3	6
1	4	5
1	4	6
1	5	6

$i = 1$

2	3	4
2	3	5
2	3	6
2	4	5
2	4	6
2	5	6

$i = 2$

3	4	5
3	4	6
3	5	6

$i = 3$

4	5	6
---	---	---

$i = 4$

```
for i=1:n
    for j=i+1:n
        for k=j+1:n
            disp([i j k])
        end
    end
end
```

Structures with array fields

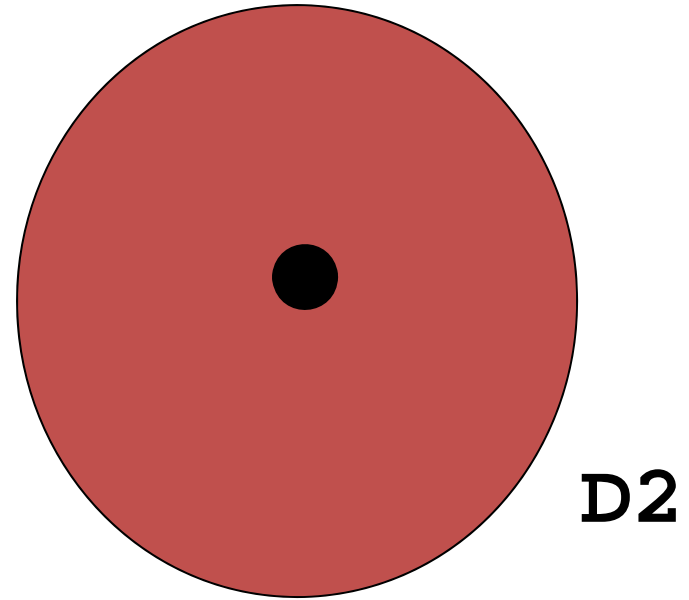
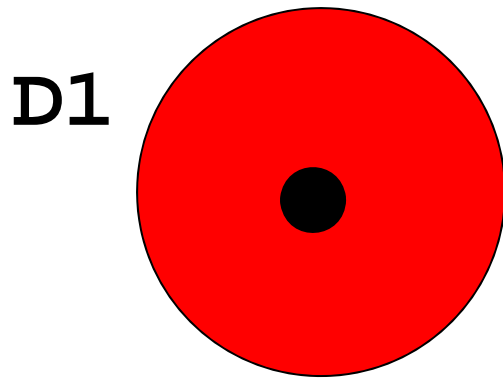
Let's develop a structure that can be used to represent a colored disk. It has four fields:

xc: x-coordinate of center
yc: y-coordinate of center
r: radius
c: rgb color vector

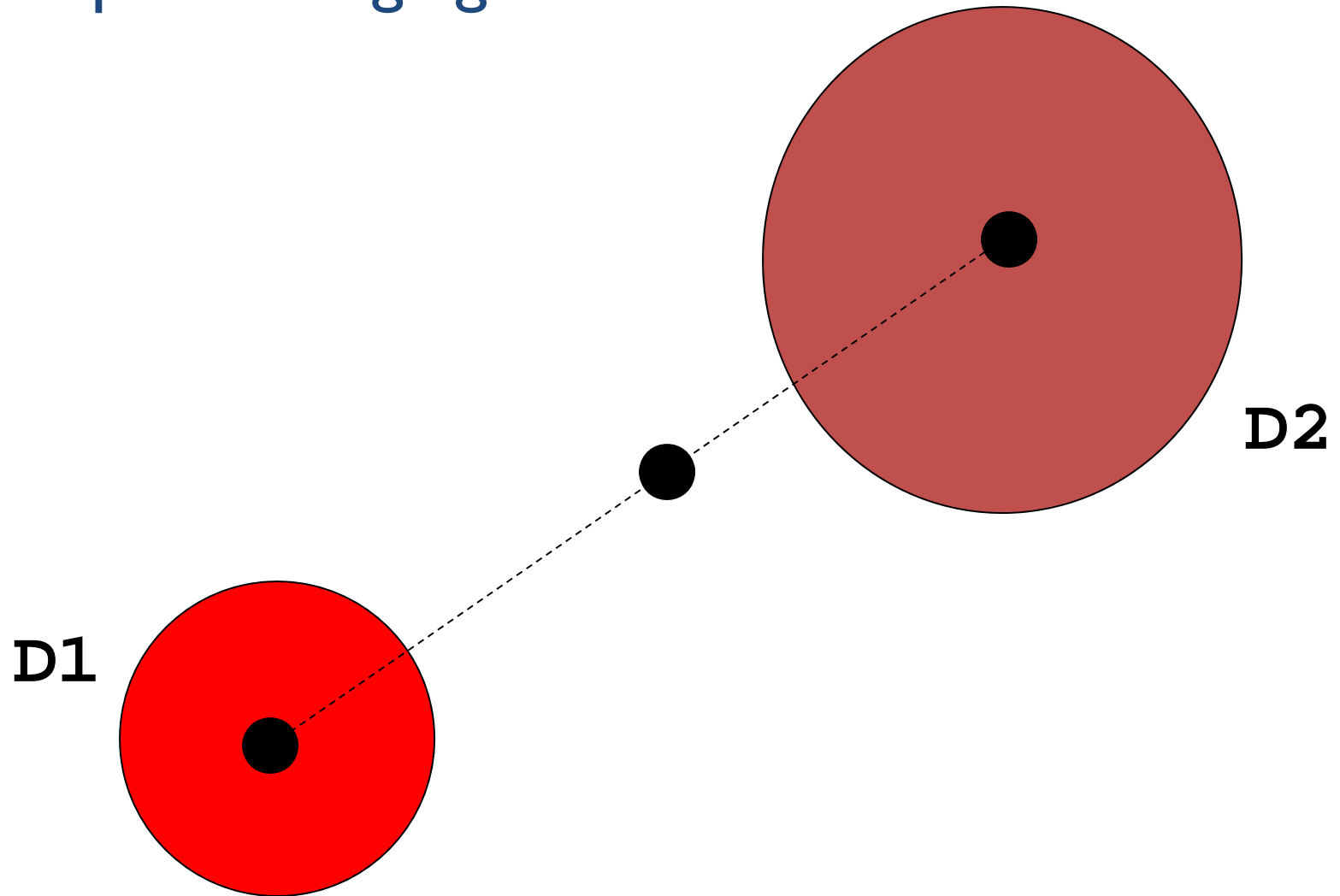
Examples:

```
D1 = struct('xc',1,'yc',2,'r',3,...  
           'c',[1 0 1]);  
D2 = struct('xc',4,'yc',0,'r',1,...  
           'c',[.2 .5 .3]);
```

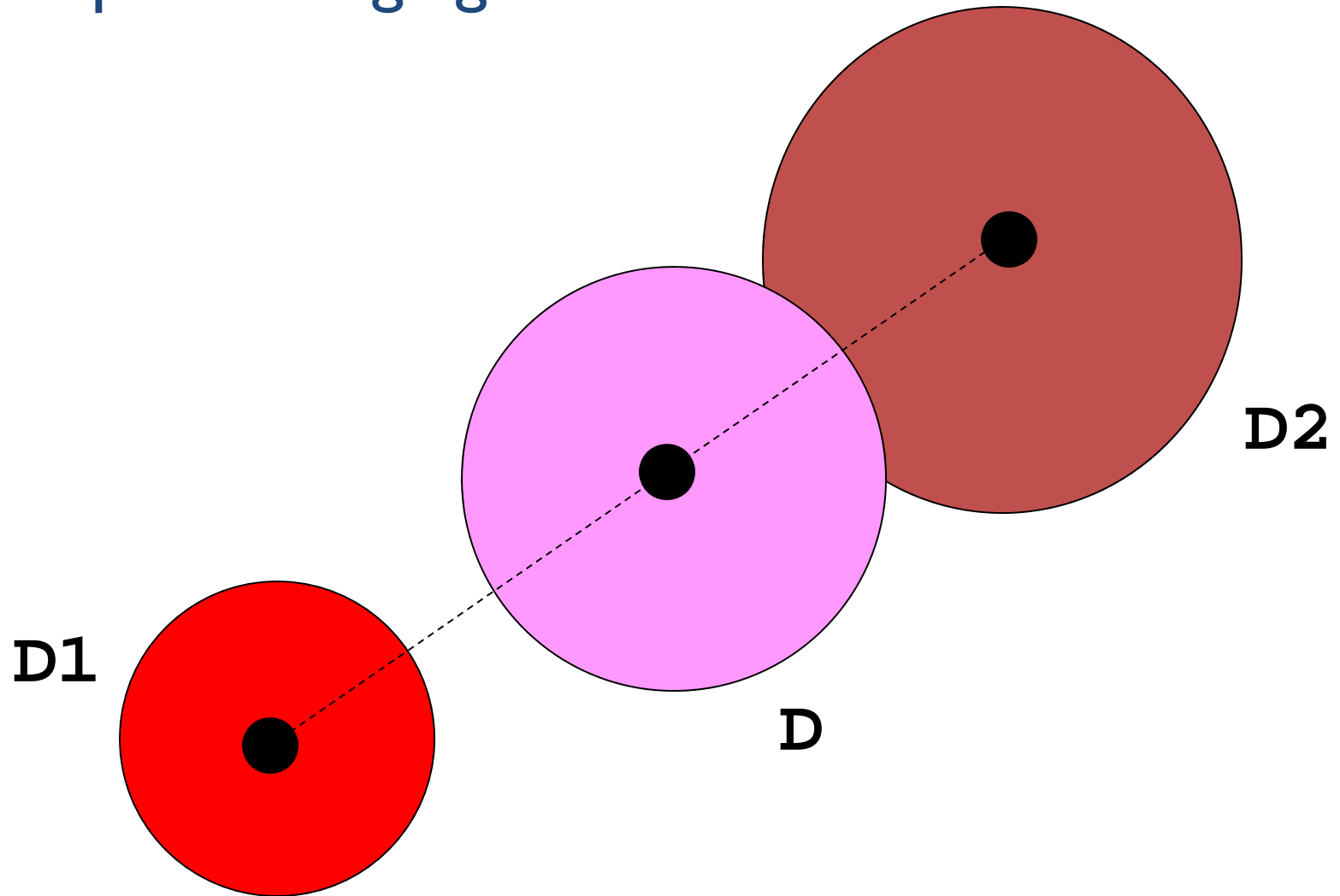
Example: Averaging two disks



Example: Averaging two disks



Example: Averaging two disks



Example: compute “average” of two disks

```
% D1 and D2 are disk structures.
```

```
% Average is:
```

```
r = (D1.r + D2.r) / 2;
```

```
xc = (D1.xc + D2.xc) / 2;
```

```
yc = (D1.yc + D2.yc) / 2;
```

```
c = (D1.c + D2.c) / 2;
```

```
% The average is also a disk
```

```
D = struct('xc',xc,'yc',yc,'r',r,'c',c)
```

How do you assign to **g** the green-color component of disk **D**?

```
D= struct('xc',3.5, 'yc',2, ...  
         'r',1.0, 'c',[.4 .1 .5])
```

A: `g = D.g;`

B: `g = D.c.g;`

C: `g = D.c.2;`

D: `g = D.c(2);`

E: *other*

A structure's field can hold a structure

```
A = MakePoint(2,3)
```

```
B = MakePoint(4,5)
```

```
L = struct('P',A,'Q',B)
```

Recall that a Point has the fields x, y

- This could be used to represent a line segment with endpoints P and Q, for instance
- Given the MakePoint function to create a point structure, what is **x** below?

```
x = L.P.y;
```

A: 2

B: 3

C: 4

D: 5

E: *error*

Different kinds of abstraction

- Packaging **procedures** (program **instructions**) into a **function**
 - A program is a set of functions executed in the specified order
 - Data is passed to (and from) each function
- Packaging **data** into a **structure**
 - Elevates thinking
 - Reduces the number of variables being passed to and from functions
- Packaging **data**, and the **instructions** that work on those data, into an **object**
 - A program is the interaction among objects
 - Object-oriented programming (OOP) focuses on the design of data-instructions groupings

A card game, developed in two ways

- Develop the algorithm—the logic—of the card game:
 - Set up a deck as an array of cards. (First, choose representation of cards.)
 - Shuffle the cards
 - Deal cards to players
 - Evaluate each player’s hand to determine winner
- Identify “objects” in the game and define each:
 - Card
 - Properties: suit, rank
 - Actions: compare, show
 - Deck
 - Property: array of Cards
 - Actions: shuffle, deal, get #cards left
 - Hand ...
 - Player ...
- Then write the game—the algorithm—using objects of the above “classes”

Procedural programming:
focus on the algorithm, i.e.,
the procedures, necessary
for solving a problem

A card game, developed in two ways

- Develop the algorithm—the logic—of the card game:
 - Set up a deck as an array of cards. (First, choose representation of cards.)
 - Shuffle the cards
 - Deal cards to players
 - Evaluate each player’s hand to determine winner

Procedural programming: focus on the algorithm, i.e., the procedures, necessary for solving a problem

- Identify “objects” in the game and define each:
 - Card
 - Properties: suit, rank
 - Actions: compare, show
 - Deck
 - Property: array of Cards
 - Actions: shuffle, deal, get #cards left
 - Hand ...
 - Player ...

- T Object-oriented programming: focus on the design of the objects (data + actions) necessary for solving a problem

Notice the two steps involved in OOP?

- Define the classes (of the objects)
 - Identify the properties (data) and actions (methods, i.e., functions) of each class
- Create the objects (from the classes) that are then used—that interact with one another

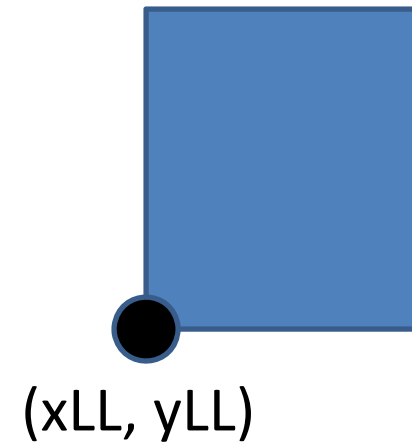
Defining a class \neq creating an object

- A class is a specification
 - E.g., a cookie cutter specifies the shape of a cookie
- An object is a concrete instance of the class
 - Need to apply the cookie cutter to get a cookie (an instance, the object)
 - Many instances (cookies) can be made using the class (cookie cutter)
 - Instances do not interfere with one another. E.g., biting the head off one cookie doesn't remove the heads of the other cookies



Example class: Rectangle

- Properties:
 - x_{LL} , y_{LL} , width, height
- Methods (actions):
 - Calculate area
 - Calculate perimeter
 - Draw
 - Intersect (the intersection between two rectangles is a rectangle!)



Example class: Time

- **Properties:**
 - Hour, minute, second
- **Methods (actions):**
 - Show (e.g., display in hh:mm:ss format)
 - Advance (e.g., advance current time by some amount)

Example class: Window (e.g., dialog box)

- Properties:
 - Title, option buttons, input dialog ...
- Methods (actions):
 - Show
 - Resize
 - ...

Many such useful classes have been predefined!

