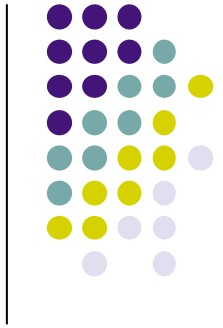
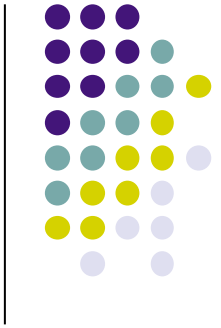
Adding abilities to objects in four steps



- Objects in OOP are much more than simple data abstractions: they add important abilities needed for practical programming

- Let us start with an object abstraction and extend it in four steps:
  - First step: a single object (data abstraction)
  - Second step: a single entry point (dispatch)
  - Third step: creating multiple objects (instantiation)
  - Fourth step: specialized syntax (classes)

First step: An object



```
declare
local
   A1={NewCell I1}
   …
   An={NewCell In}
in
   proc {M1 …} … end
   …
   proc {Mm …} … end
end
```

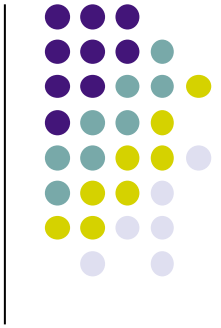
This code gives the structure of an object abstraction.

An object is a combination of local cells A1, …, An and global procedures M1, …, Mm.

We call A1, …, An the "attributes" and M1, …, Mm the "methods".

Attributes A1, ..., An are *hidden* from the outside and methods M1, ..., Mm are *visible* from the outside (interface!).

First step: An object



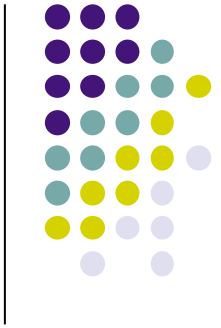
```
declare
local
   A1={NewCell 0}
in
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
end

{Inc}
local X in {Get X} {Browse X} end
```

This code creates one object that implements a counter.

The object has two methods, Inc and Get, and is initialized to 0.

Since the cell can only be accessed with the methods, the behavior is guaranteed correct: {Get X} binds X to an integer that gives the number of calls {Inc} done before.

Second step: Single entry point



```
declare
local
   A1={NewCell 0}
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
in
   proc {Counter M}
      case M of inc then {Inc}
      [] get(X) then {Get X}
      end
   end
end
```

This extends the counter object to invoke all methods from a single entry point: the procedure Counter.

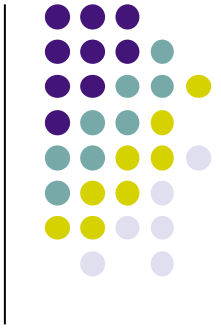
{Counter inc}

{Counter inc}

{Counter get(X)}

In this example, this is called procedure dispatch, since the entry point is a procedure. The argument M is usually called a message.

Third step: Creating multiple objects



```
declare
fun {NewCounter}
   A1={NewCell 0}
   proc {Inc} A1:=@A1+1 end
   proc {Get X} X=@A1 end
in
   proc {$ M}
      case M of inc then {Inc}
      [] get(X) then {Get X}
      end
   end
end
```

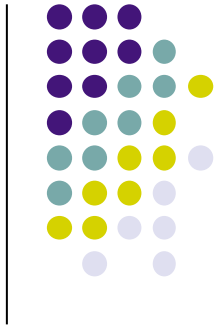
We add the ability to create many counter objects with the same methods but different states.

The function NewCounter creates a new counter object each time it is called. This is an example of instantiation (higher-order programming).

The call C={NewCounter} creates a new cell in A1 and returns an object with methods Inc and Get, that both access the new cell.

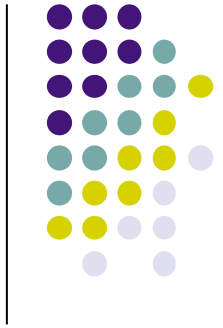
Each new object is completely independent of the others.

Using NewCounter



```
C1={NewCounter} % First object
C2={NewCounter} % Second object

{C1 inc} % Increment first object twice
{C1 inc}

local X in {C1 get(X)} {Browse X} end % Shows 2
local X in {C2 get(X)} {Browse X} end % Shows 0
```

Fourth step: Specialized syntax



```
class Counter
   attr a1
   meth init a1:=0 end
   meth inc a1:=@a1+1 end
   meth get(X) X=@a1 end
end

C1={New Counter init}
{C1 inc}
local X in
   {C1 get(X)} {Browse X}
end
```

We introduce a new syntax for defining objects, in which we define attributes and methods.

We call this definition a class, since we can use it to define many objects with the same behavior (they are of the same class). We separate the object definition (the class) from the object creation (the function New).

The new syntax guarantees that the object is constructed without error. It also improves readability and lets the system improve performance.