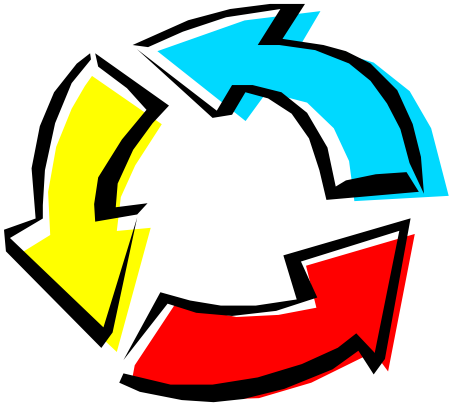
Recursion

Lab 09 and 10

Recursion



Recursion occurs when a method calls itself.

Recursion

```
public class RecursionOne
{
  public void run(int x)
  {
    out.println(x);
    run(x+1);
  }
  public static void main(String args[]  )
  {
    RecursionOne test = new RecursionOne();
    test.run(1);
  }
}
```

Will it stop?

OUTPUT

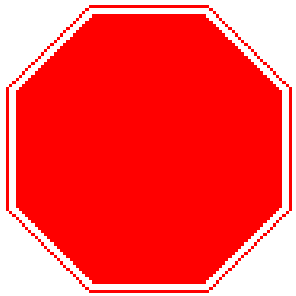
1
2
3
4
5
. . . . .
stack overflow

open recursionone.java

Base Case

A recursive method must have a stop condition/ base case.

Recursive calls will continue until the stop condition is met.



Recursion 2

```
public class RecursionTwo
{
  public void run(int x )
  {
    out.println(x);
    if(x<5)
      run(x+1);
  }
  public static void main(String args[]  )
  {
    RecursionTwo test = new RecursionTwo();
    test.run(1);
  }
}
```

base case
It will stop!

OUTPUT

1
2
3
4
5

Recursion 3

```
public class RecursionThree
{
  public void run(int x )
  {
    if(x<5)
      run(x+1);
    out.println(x);
  }
  public static void main(String args[]  )
  {
    RecursionThree test = new RecursionThree ();
    test.run(1);
  }
}
```

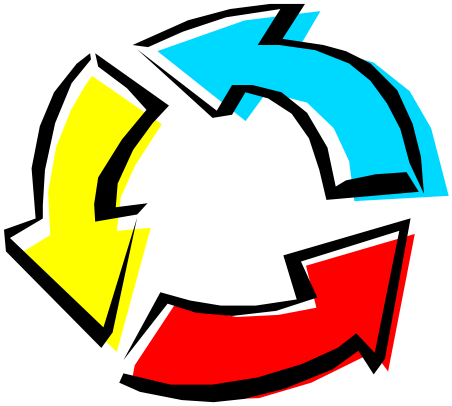
base case

OUTPUT

5
4
3
2
1

open recursiontwo.java recursionthree.java

Recursion



Recursion is basically a loop that is created using method calls.

do while

```
class DoWhile
{
  public void run( )
  {
    int x=0;
    do{
      x++;
      out.println(x);
    }while(x<10);        //condition
  }
  public static void main(String args[]  )
  {
    DoWhile test = new DoWhile();
    test.run( );
  }
}
```

open dowhile.java

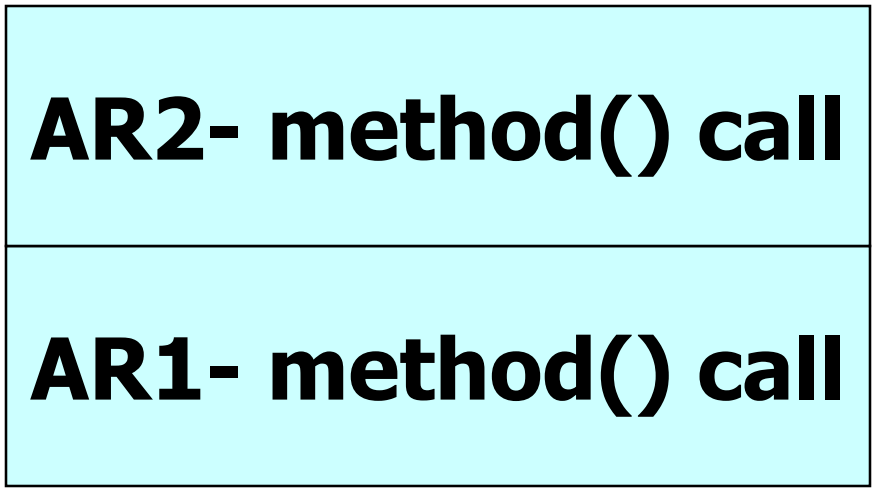
The Stack

When you call a method, an activation record for that method call is put on the stack with spots for all parameters/arguments being passed.

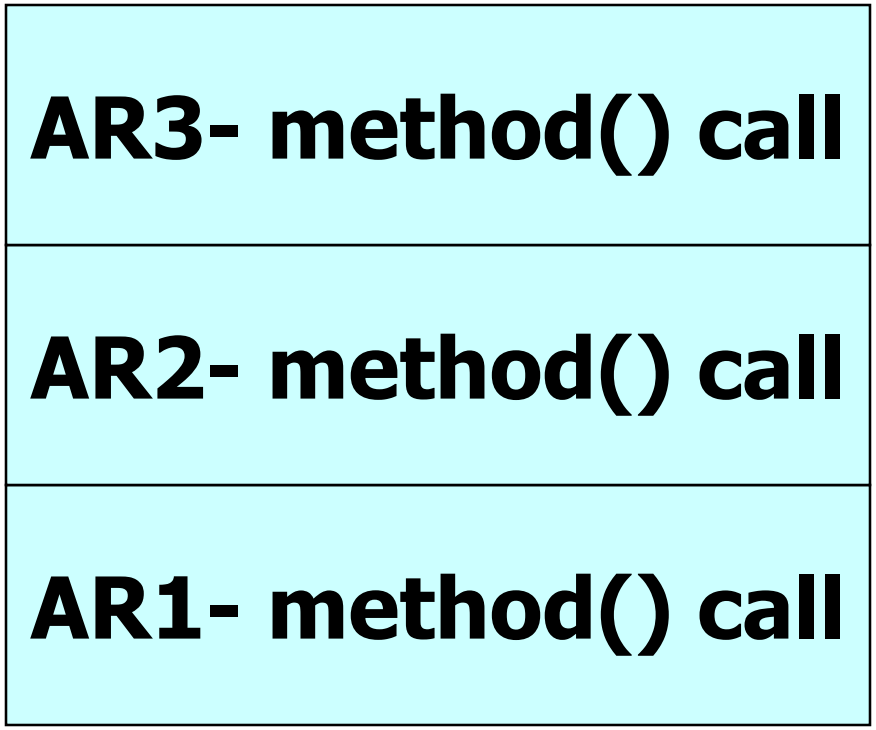
The Stack

AR1- method() call

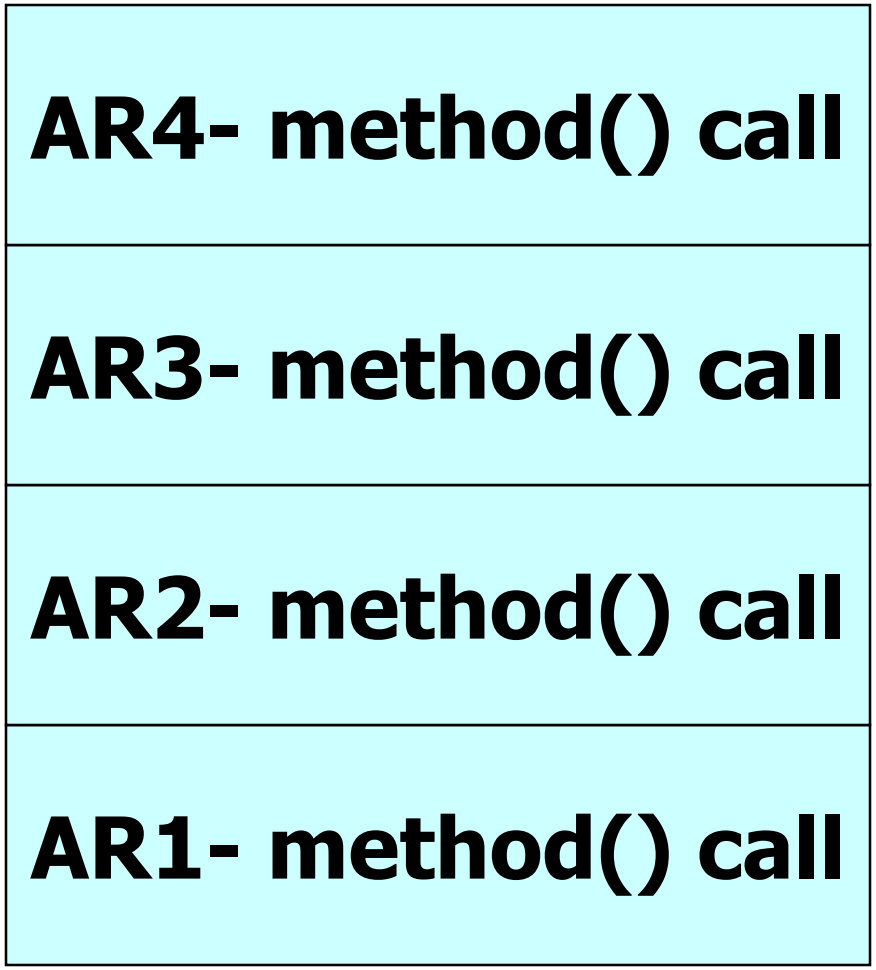
The Stack



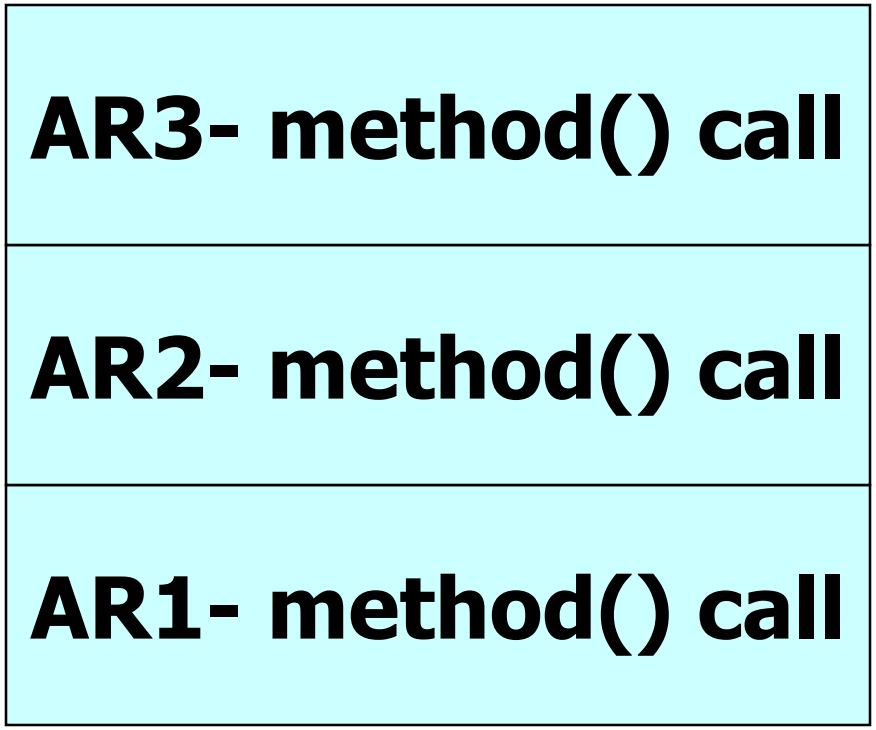
The Stack



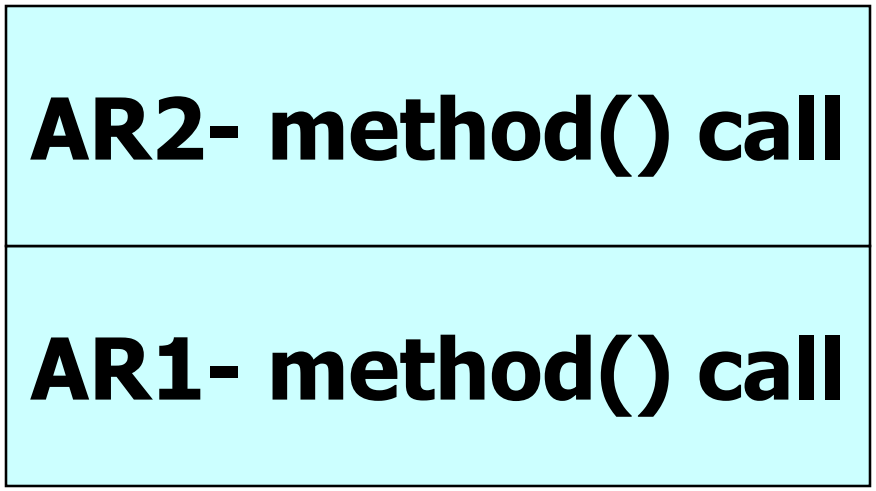
The Stack



The Stack



The Stack



The Stack

As each call to the method completes, the instance of that method is removed from the stack.

AR1- method() call

Recursion 2

```
public class RecursionTwo
{
  public void run(int x )
  {
    out.println(x);
    if(x<5)
      run(x+1);
  }
  public static void main(String args[]  )
  {
    RecursionTwo test = new RecursionTwo();
    test.run(1);
  }
}
```

base case
It will stop!

OUTPUT

1
2
3
4
5

Recursion 3

```
public class RecursionThree
{
  public void run(int x )
  {
    if(x<5)
      run(x+1);
    out.println(x);
  }
  public static void main(String args[]  )
  {
    RecursionThree test = new RecursionThree();
    test.run(1);
  }
}
```

base case

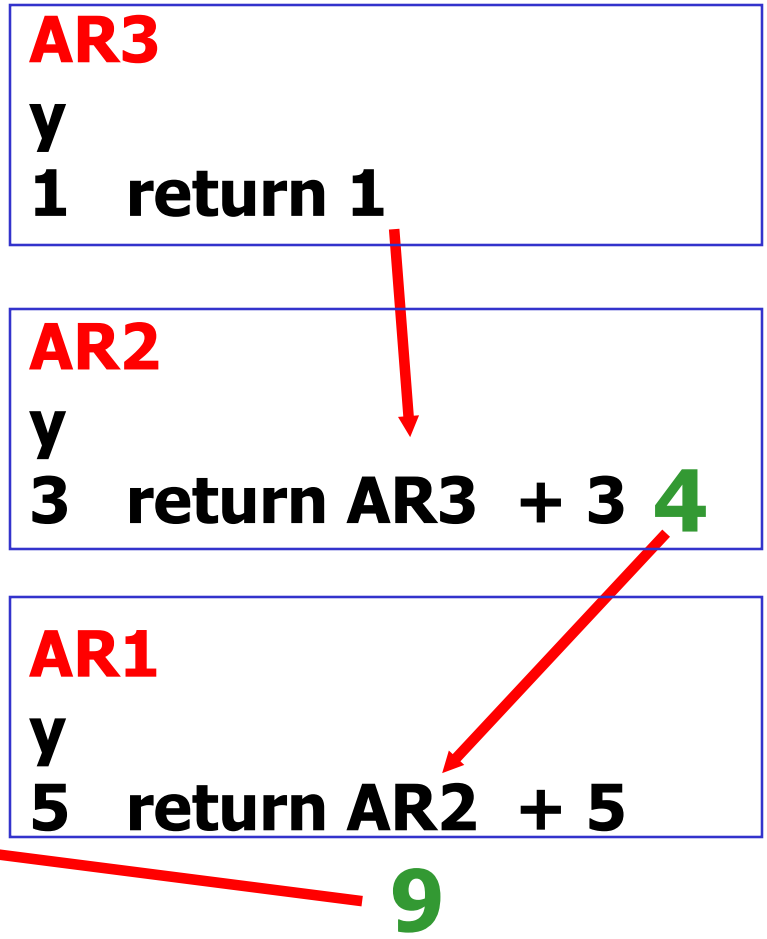
OUTPUT

5
4
3
2
1

Why does this output differ from recur2?

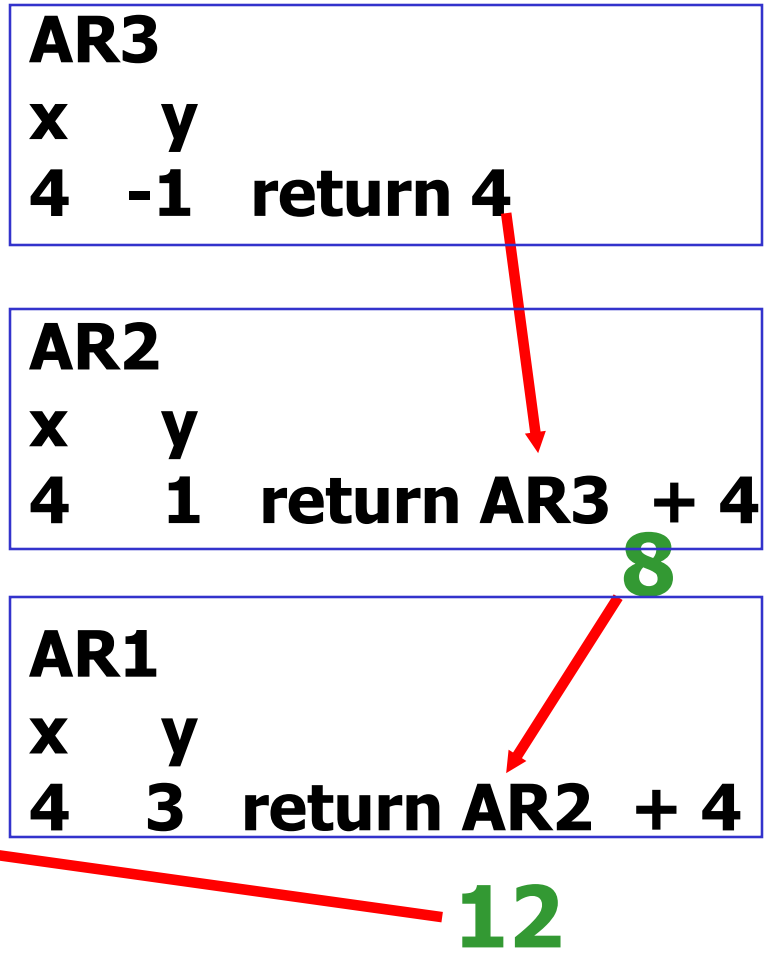
Tracing Recursive Code

```
int fun(int y)
{
  if(y<=1)
    return 1;
  else
    return fun(y-2) + y;
}

//test code in client class
out.println(test.fun(5));
```



Tracing Recursive Code

```
int fun( int x, int y)
{
  if( y < 1)
     return x;
  else
     return fun( x, y - 2) + x;
}

//test code in client class
out.println(test.fun(4,3));
```



open recursionfour.java recursionfive.java

Recursive Fun

```
int fun(int x, int y)
{
   if ( x == 0 )
      return x;
   else
      return x+fun(y-1,x);
}
```

OUTPUT

16

What would fun(4,4) return?

open recursionsix.java

split recursion

tail recursion

```
public String recur(String s)
{
  int len = s.length();
  if(len>0)
       return recur(s.substring(0,len-1)) +
                                        s.charAt(len-1);
  return "";
}
```

split recursion

tail recursion

```
public String recur(String s)
{
   int len = s.length();
   if(len>0)
        return s.charAt(len-1) +
                              recur(s.substring(0,len-1));
   return "";
}
```

open recursionseven.java

recursioneight.java

split recursion

tail recursion

The Stack

call out.println(recur("abc"))

```
public String recur(String s)
{
  int len = s.length();
  if(len>0)
       return recur(s.substring(0,len-1)) +
                                        s.charAt(len-1);
  return "";
}
```

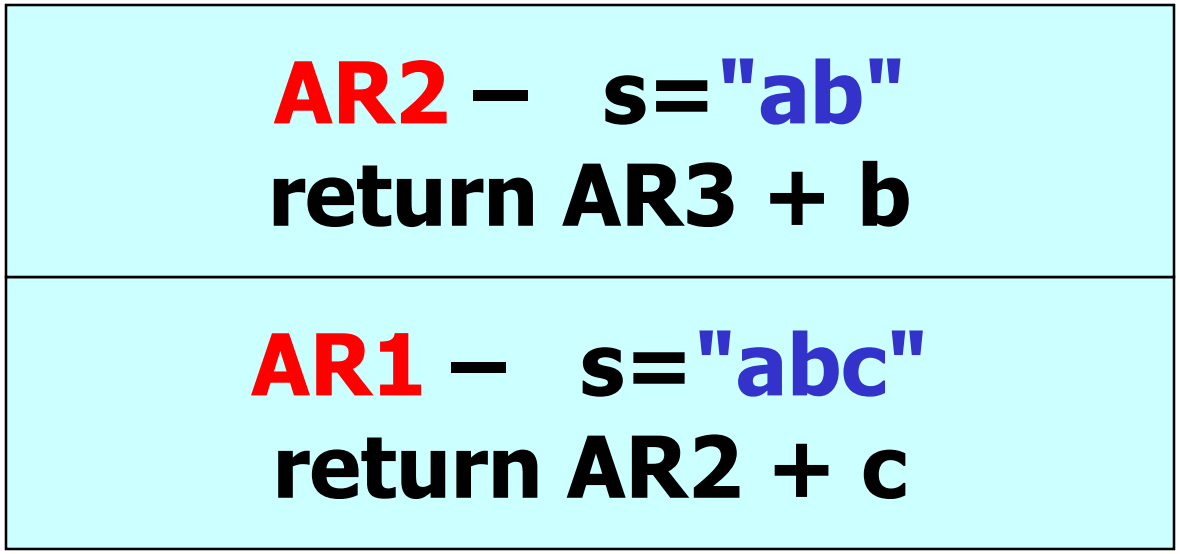
The Stack

call out.println(recur("abc"))

AR stands for activation record. An **AR** is placed on the stack every time a method is called.

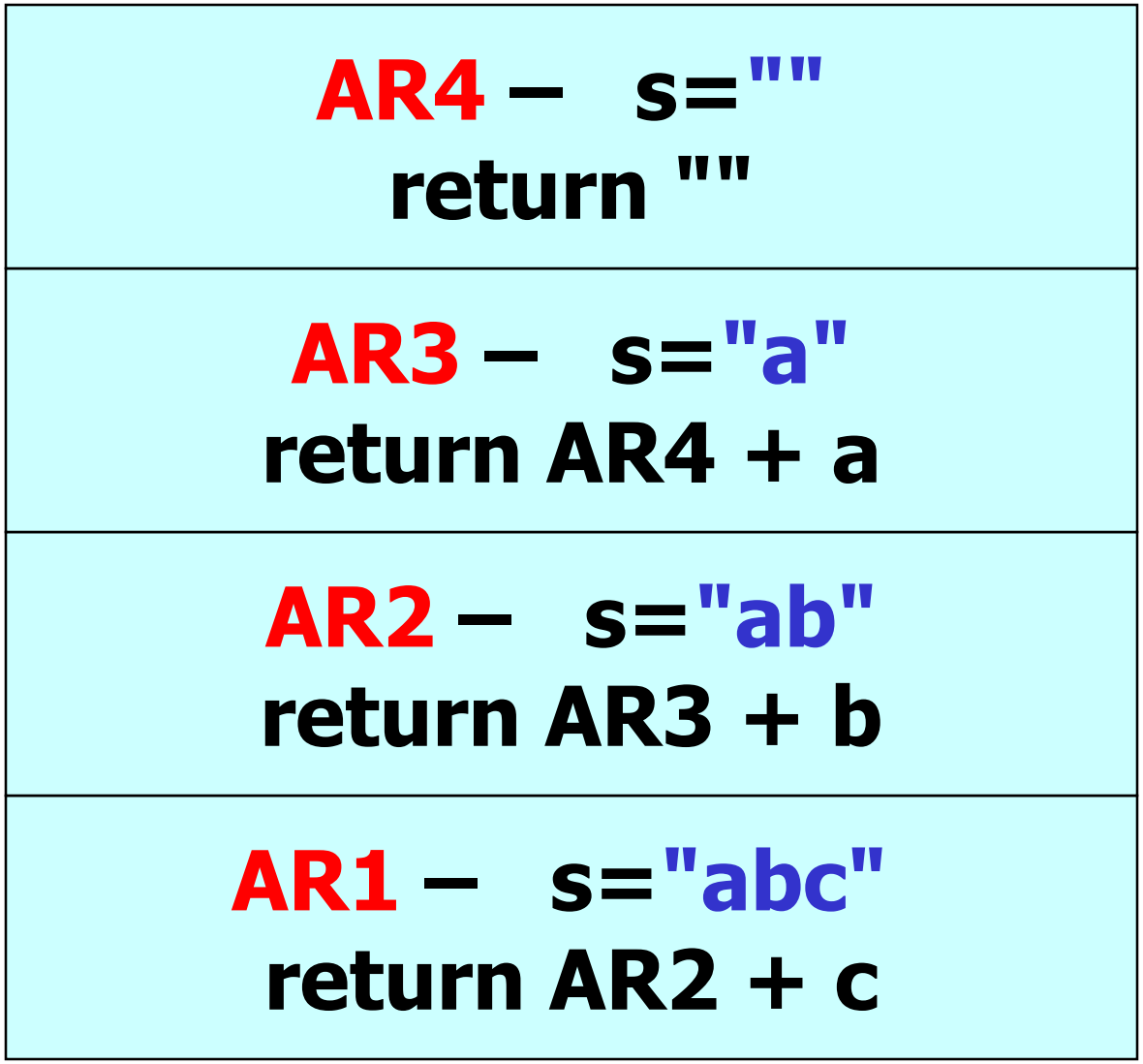
AR1 – s="abc"
return AR2 + c

The Stack

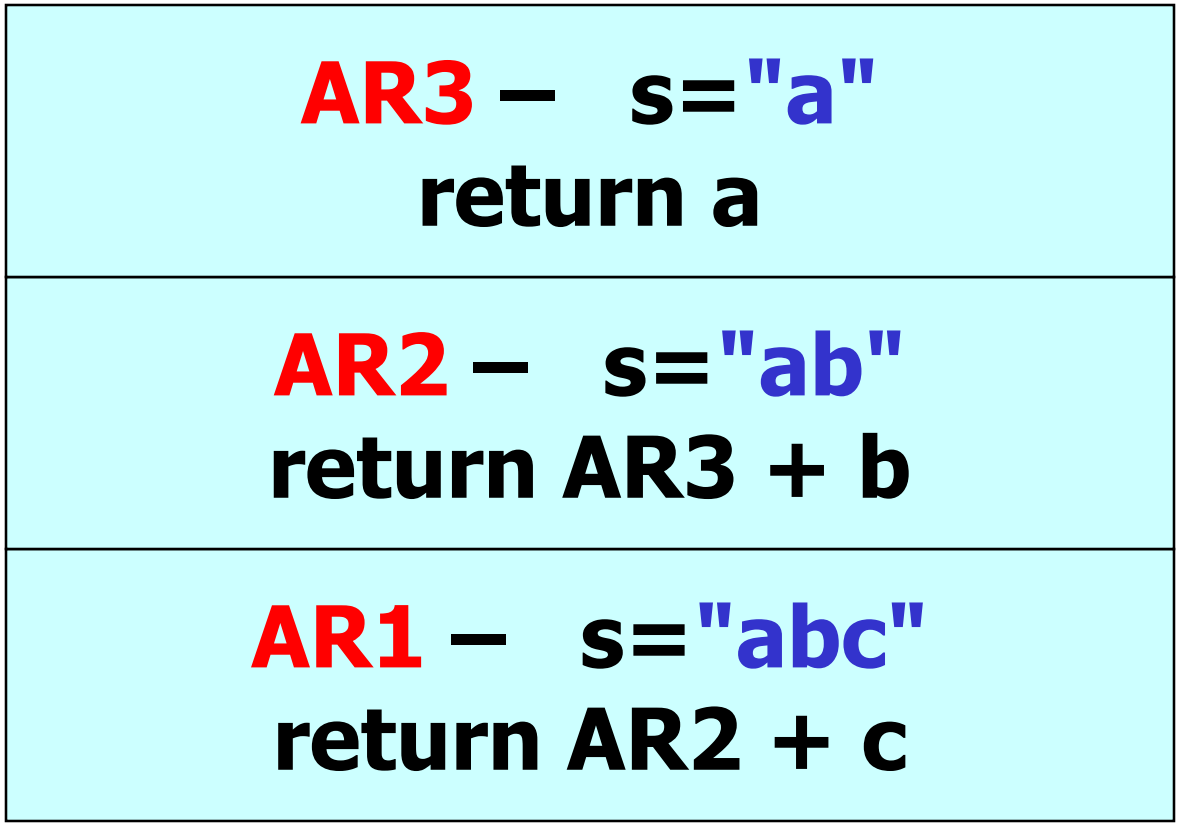


The Stack

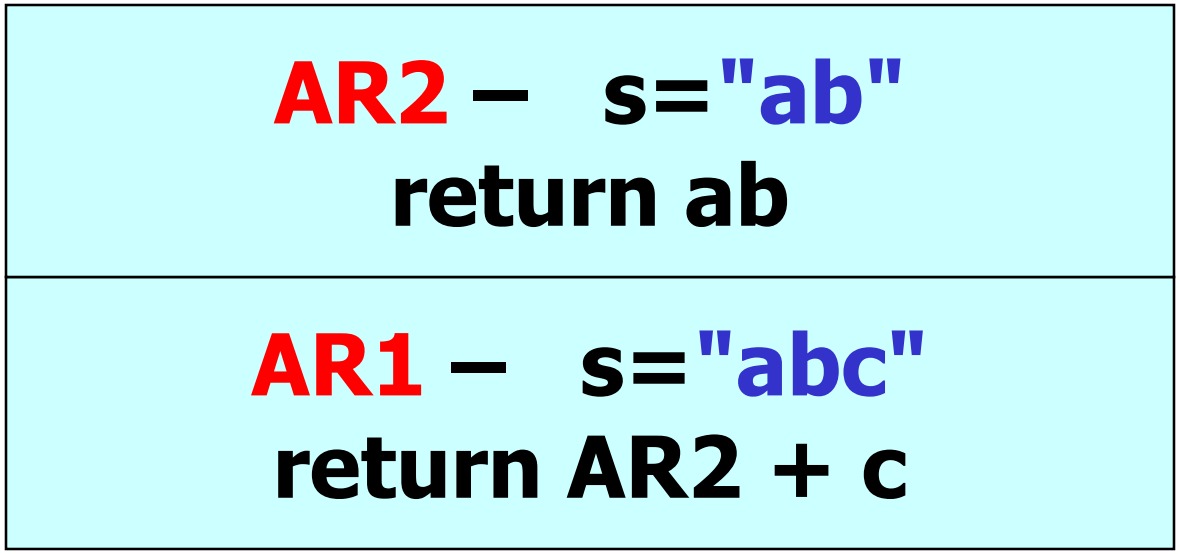
The Stack



The Stack



The Stack



The Stack

call out.println(recur("abc"))

OUTPUT

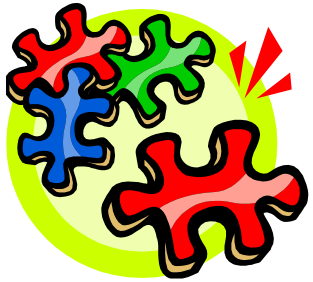
abc

AR1 – s="abc"
return abc

What is the point?

If recursion is just a loop, why would you just not use a loop?

Recursion is a way to take a block of code and spawn copies of that block over and over again. This helps break a large problem down into smaller pieces.



Counting Spots

If checking 0 0, you would find 5 @s are connected.

```
@ - @ - - @ - @ @ @
@ @ @ - @ @ - @ - @
- - - - - - - @ @ @
- @ @ @ @ @ - @ - @
- @ - @ - @ - @ - @
@ @ @ @ @ @ - @ @ @
- @ - @ - @ - - - @
- @ @ @ - @ - - - -
- @ - @ - @ - @ @ @
- @ @ @ @ @ - @ @ @
```

@ at spot [0,0]
@ at spot [0,2]
@ at spot [1,0]
@ at spot [1,1]
@ at spot [1,2]

The exact same checks are made at each spot.

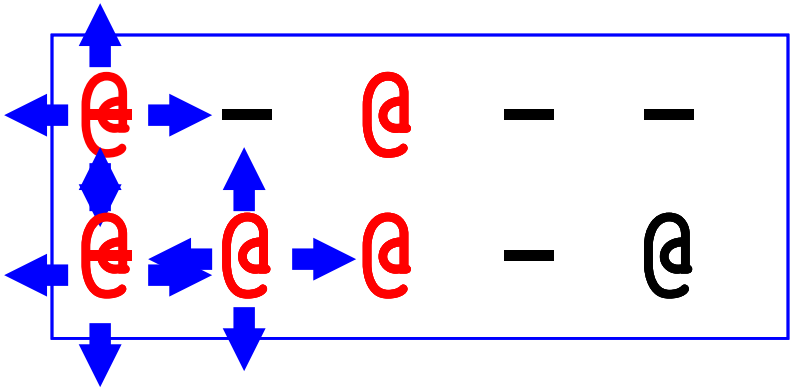
Counting Spots

```
if ( r and c are in bounds and
                        current spot is a @ )
   mark spot as visited
   bump up current count by one
   recur up
   recur down
   recur left
   recur right
```

This same block of code is recreated with each recursive call. The exact same code is used to check many different locations.

Counting Spots

```
if ( r and c are in bounds and
                          current spot is a @ )
   mark spot as visited
   bump up current count by one
   recur up
   recur down
   recur left
   recur right
```



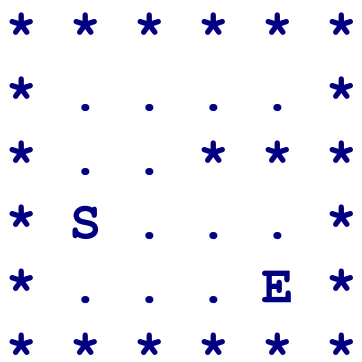
Start work on Lab 09

Advanced Recursion

Mazes

Mazes

Maze problems are very common programming problems.



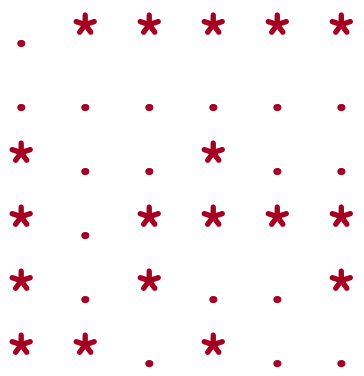
In some cases, you are provided with a symbol for the start and a symbol for the exit.

. are paths

* are walls

Mazes

Maze problems are very common programming problems.



. are paths

* are walls

Other times, you start at 0,0 and must check to see if you can get to length-1, length-1.

Maze Solving

```
void search(int row, int col)
{
    if (row >= 0 && col >= 0 && row < maze.length &&
            col < maze[r].length && maze[row][col] = = 1)
  {
    if (col = = maze[r].length - 1)  {
            exitFound = true;
    }
    else  {
            maze[row][col] = 0;   //marking
            search(row+1, col);
            search(row-1, col);
            search(row, col+1);
            search(row, col-1);
          }
   }
}
```

1	0	1	1	1
0	1	1	1	1
1	1	1	1	0
0	0	0	0	1
0	0	1	1	0

Marking / UnMarking

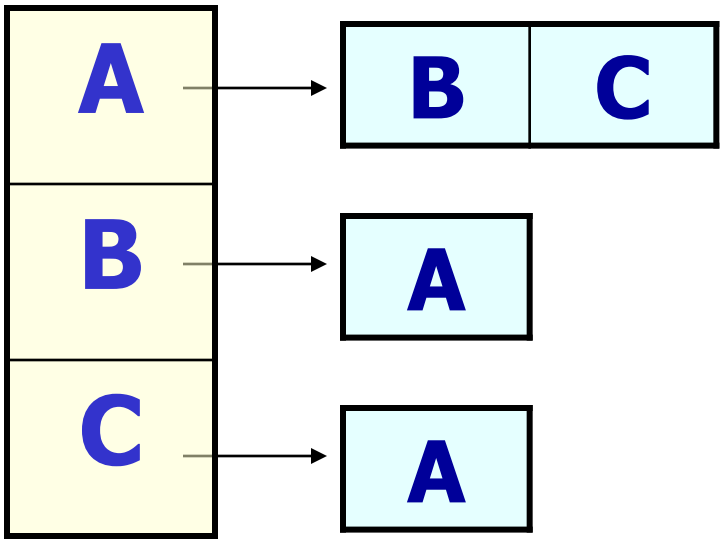
In some situations, you want to make changes to the maze temporarily. For instance, if you are trying to determine the shortest path, you have to find all paths and determine which path is the shortest. Each time to you search the maze for a path, the maze must be in its original state.

Marking / UnMarking

```
void search(int row, int col)
{
   if (row >= 0 && col >= 0 && row < maze.length &&
            col < maze[r].length && maze[row][col] = = 1)
   {
         if (col = = maze[r].length - 1)  {
           exitFound = true;
         }
         else  {
           maze[row][col] = 0;   //marking
           search(row+1, col);
           search(row-1, col);
           search(row, col+1);
           search(row, col-1);
           maze[row][col] = 1;   //unmarking ( set it back )
        }
    }
}
```

Connections

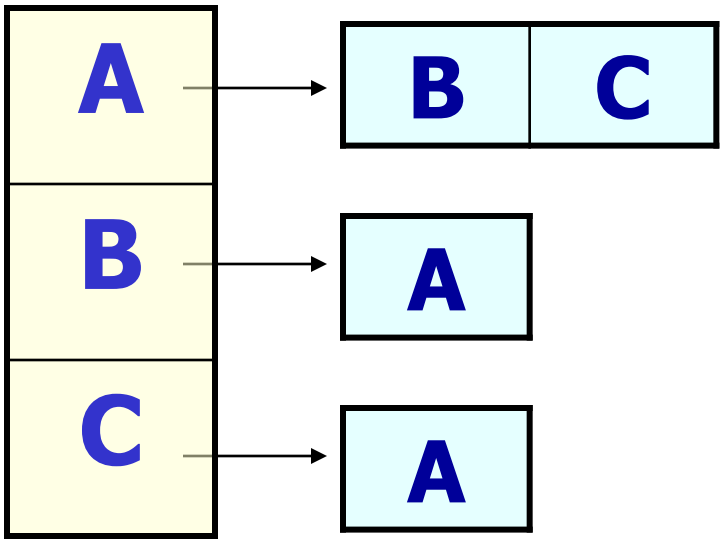
Connections



Connection problems require you to check for a path between 2 items.

Is A directly connected to C? YES

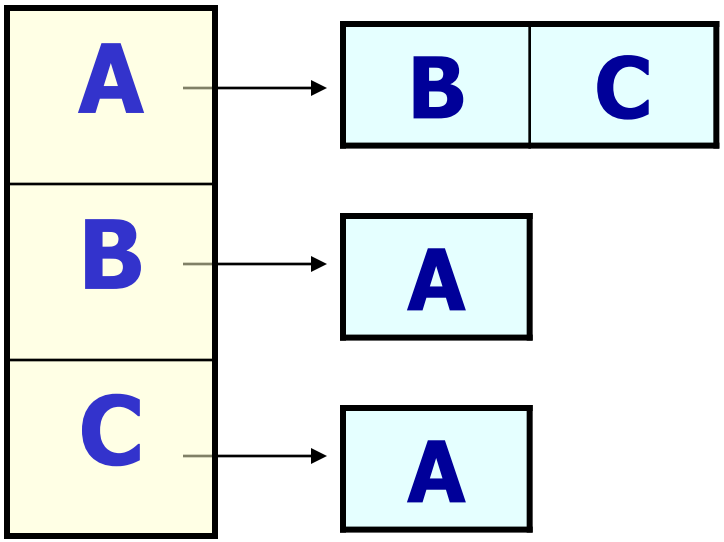
Connections



Connection problems require you to check for a path between 2 items.

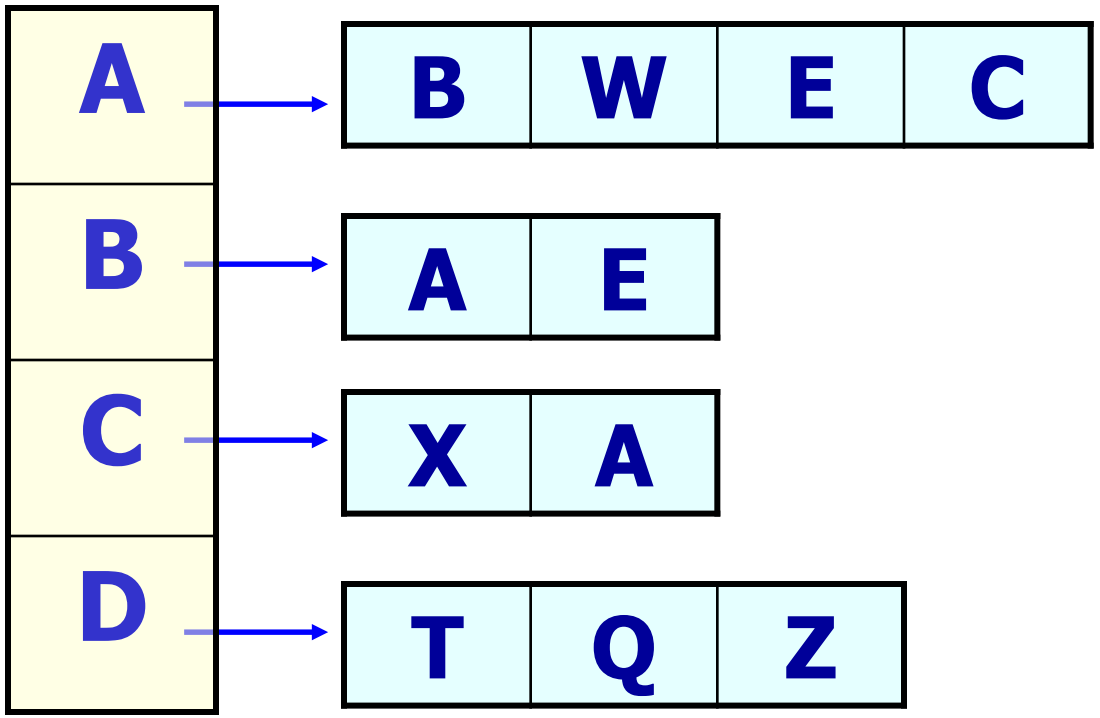
Is B directly connected to C? NO

Connections



A is directly connected to B and C.
B is directly connected to A.
C is directly connected to A.
Is B connected to C? YES

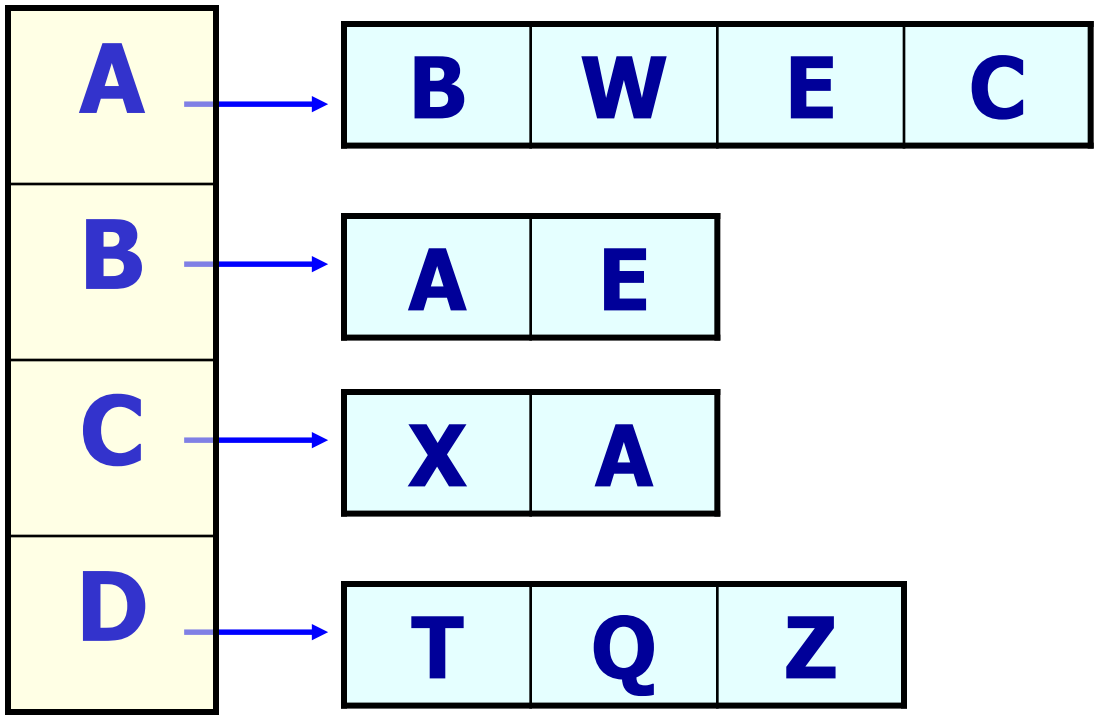
Connections



Is A connected to X? YES

Is A connected to Q? NO

Connections



TreeMap<Character, String> map;

TreeMap<Character, Set> map;

Connections

```
check(String one,  String two, String list)
{
  if a direct connection exists between one and two
       we have a match
  else
  {
     get the current list of connections for one
     loop through all of the connections
      if you have not checked the current spot
        add current spot to list
        check to see a connection exists between spot
                                  and the destination ( recursive call )
  }
}
```

Start work on Lab 10