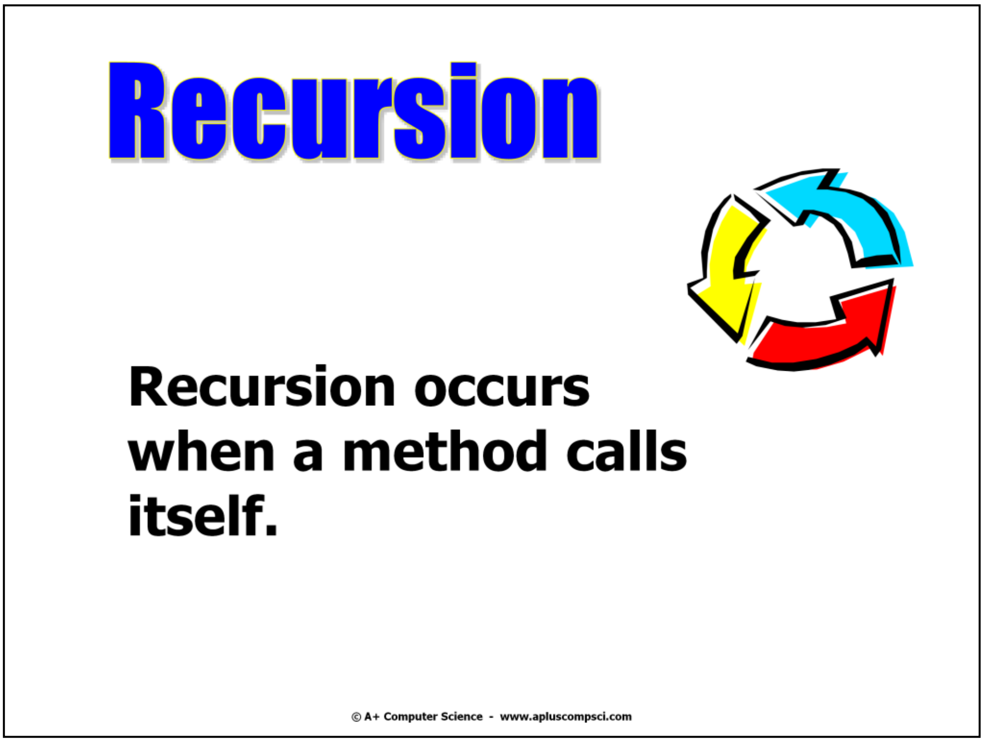
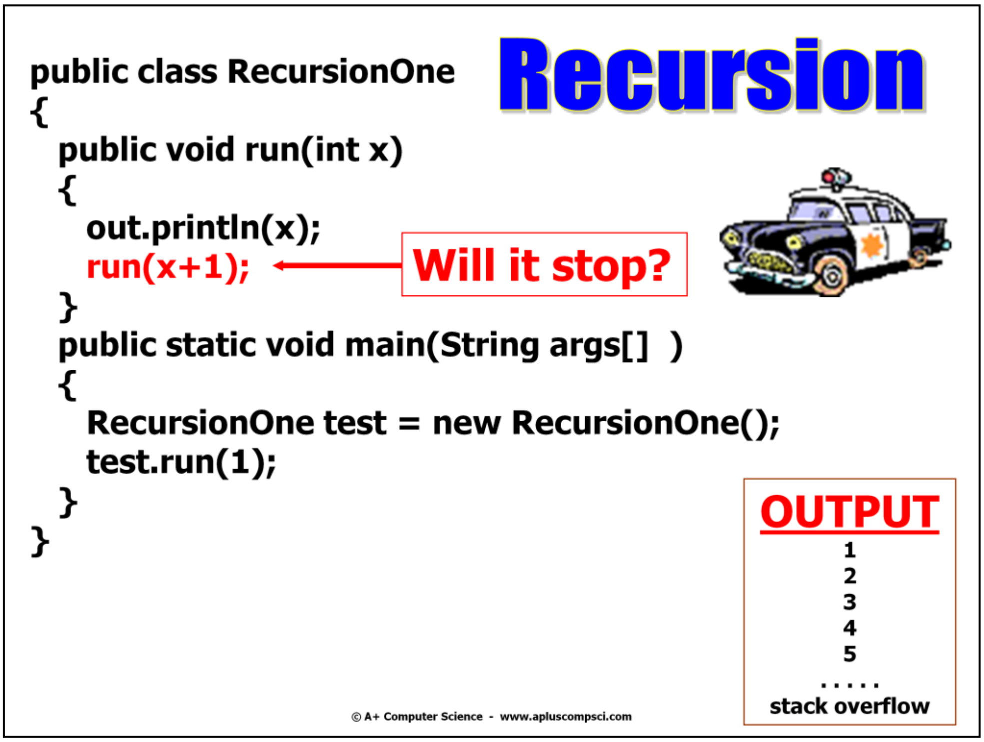
Recursion



If a method contains a call to itself, that method is recursive. Recursion is a very useful programming tool if used properly.



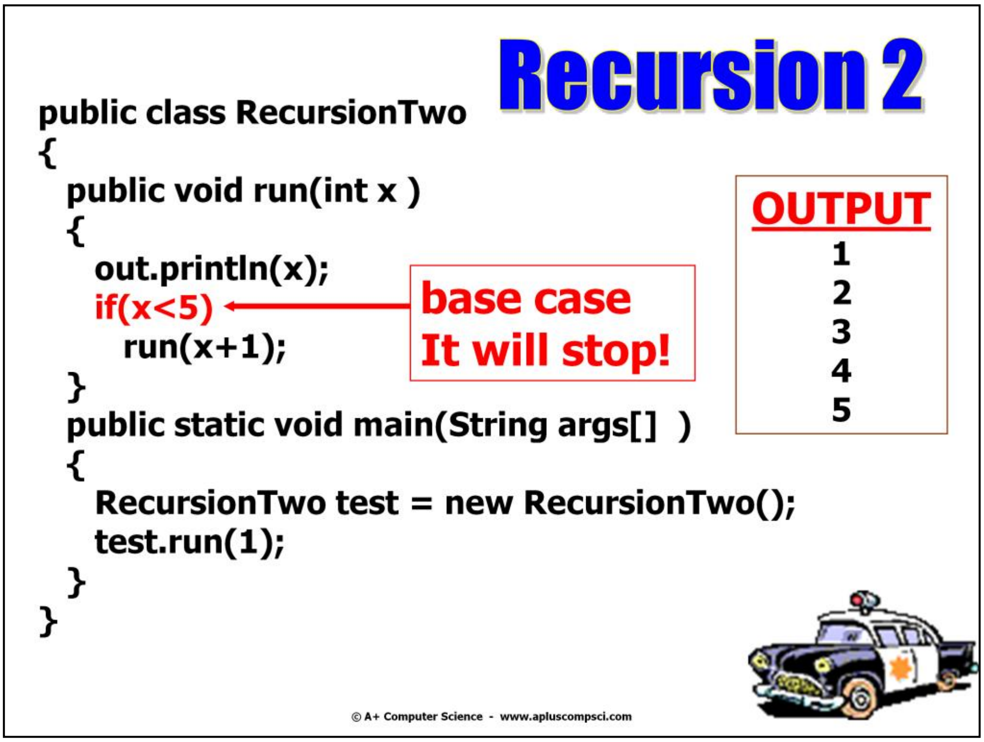
The recursive method `run()` will stop recurring when it runs out of memory. There is no code or case to make the recursion stop.

open
recursionone.java

Base Case

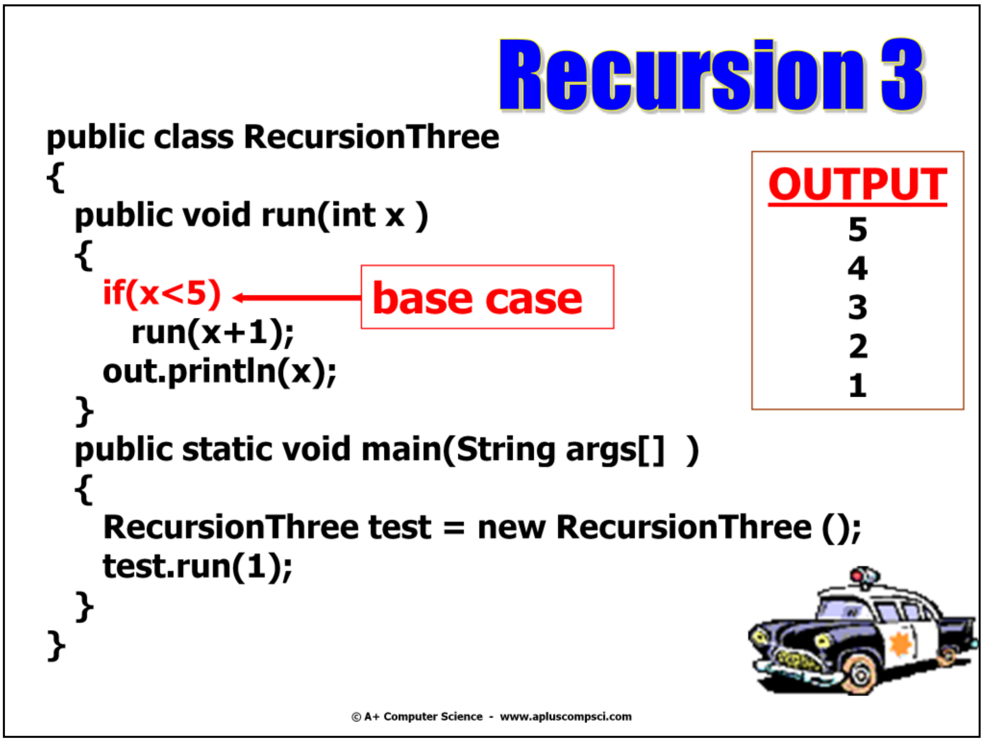
A recursive method must have a stop condition/ base case.

Recursive calls will continue until the stop condition is met.



Method `run()` has been improved as it now contains a `case(x<5)` that will prevent the recursion form going on to infinity.

The `println(x)` happens before the recursive call so the numbers appear in order.



Method `run()` has been improved as it now contains a `case(x<5)` that will prevent the recursion form going on to infinity.

The `println(x)` happens after the recursive call so the numbers appear in reverse order.

open
recursiontwo.java
recursionthree.java

Recursion

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

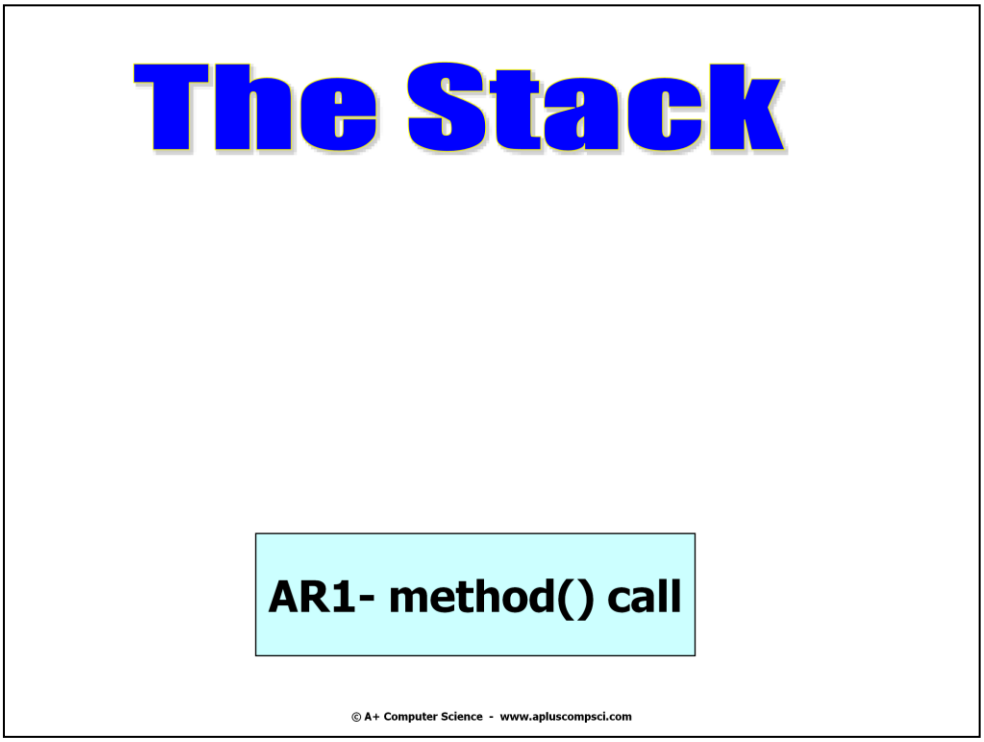
do while

```
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( );
  }
}
```

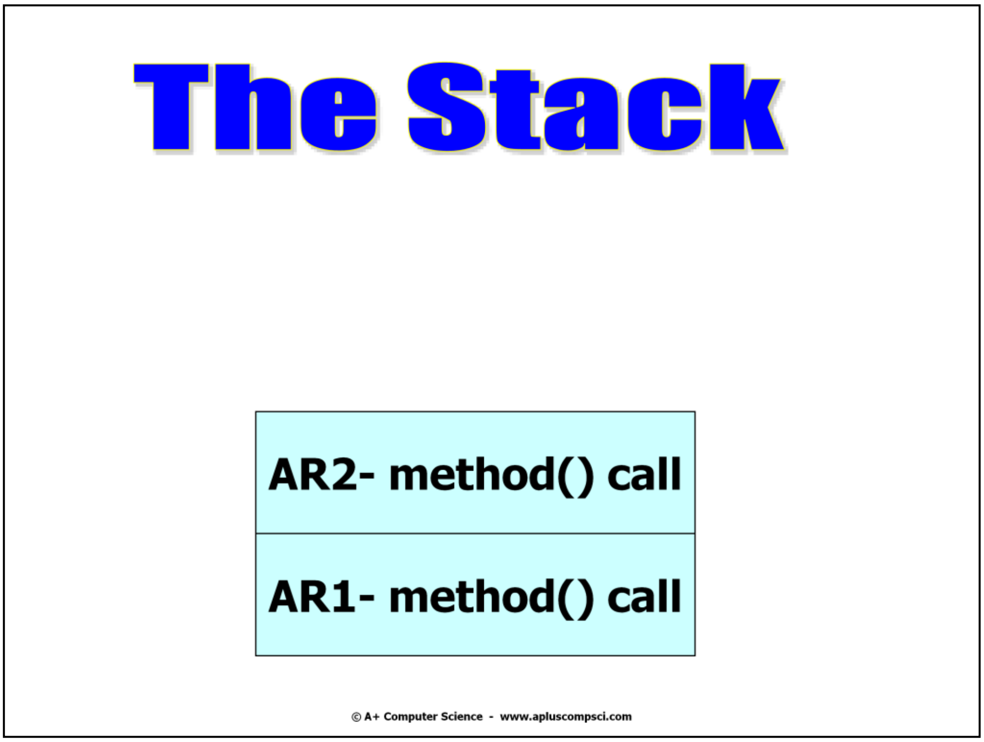
open dowhile.java

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.



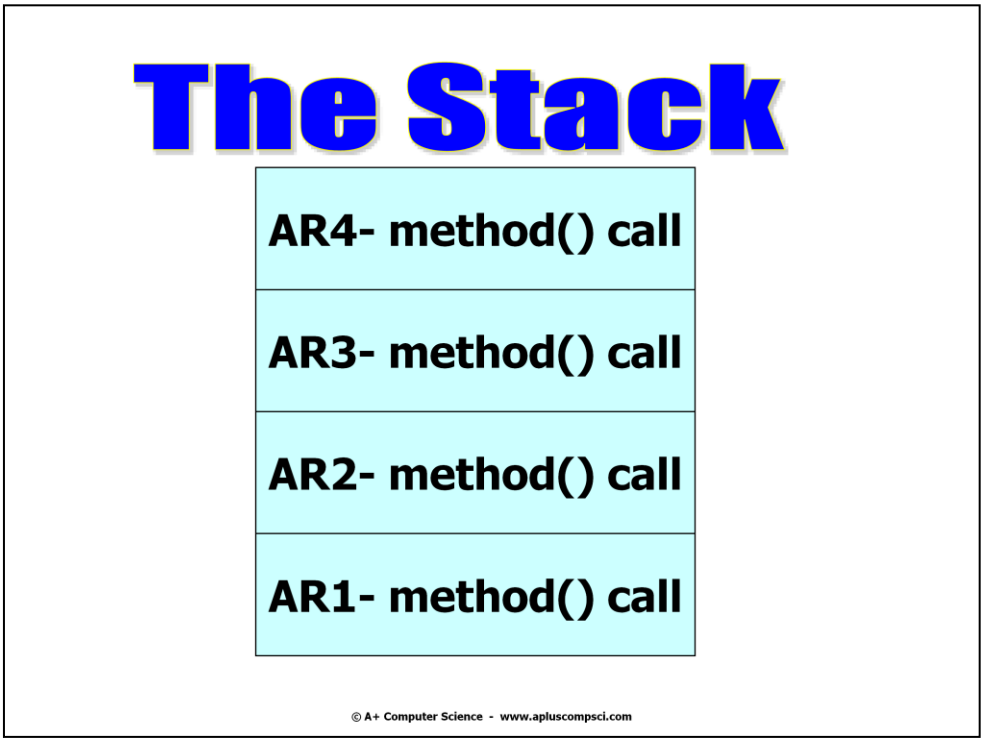
Because AR1 is placed on the stack first, it will be the last AR removed from the stack.



Because AR2 is placed on the stack second, it will be the second to last AR removed from the stack.

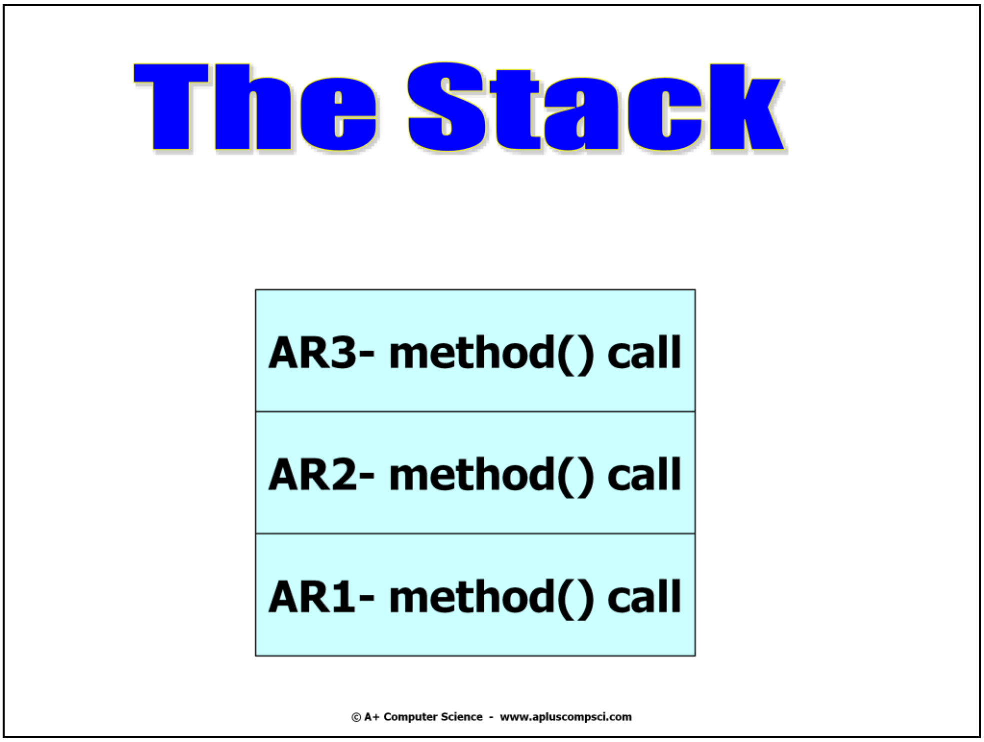
The Stack

AR3- method() call
AR2- method() call
AR1- method() call



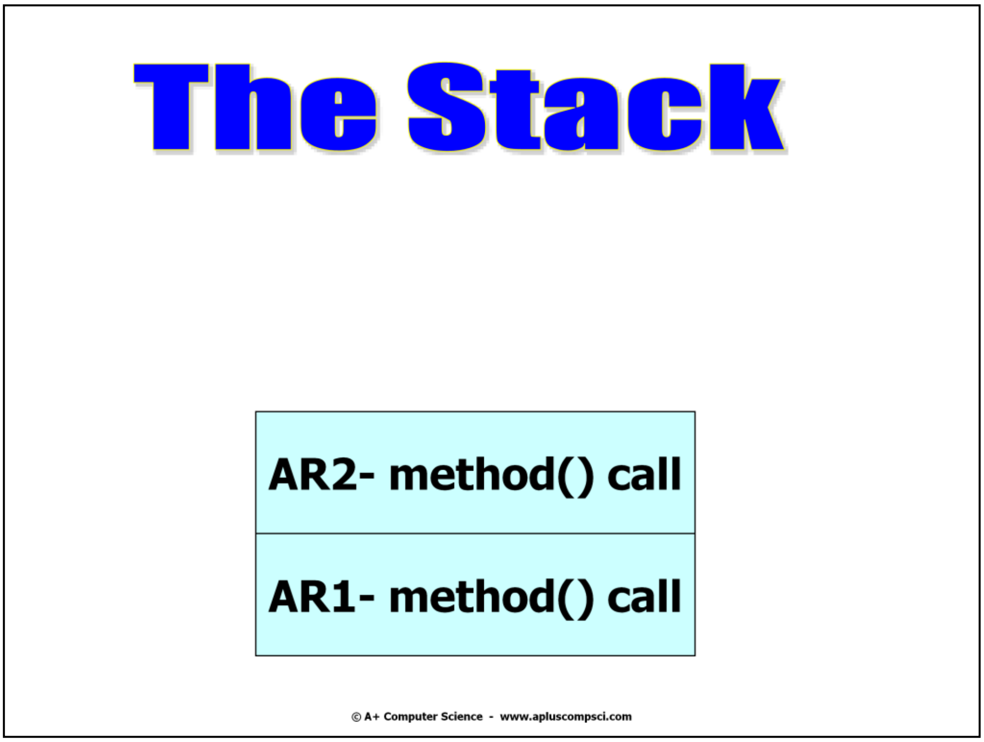
AR4 is placed on the stack last and it is processed to completion first.

Once AR4 is finished, the execution sequence returns to AR3.

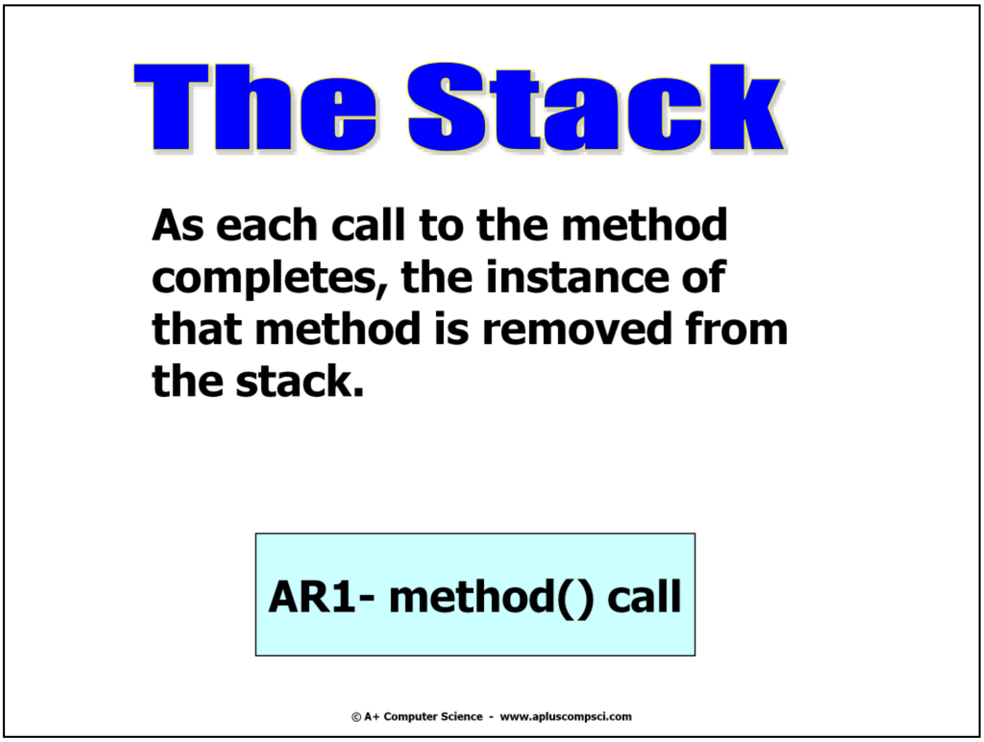


AR3 was placed on the stack 2nd to last and it is processed to completion after AR4 and before AR2.

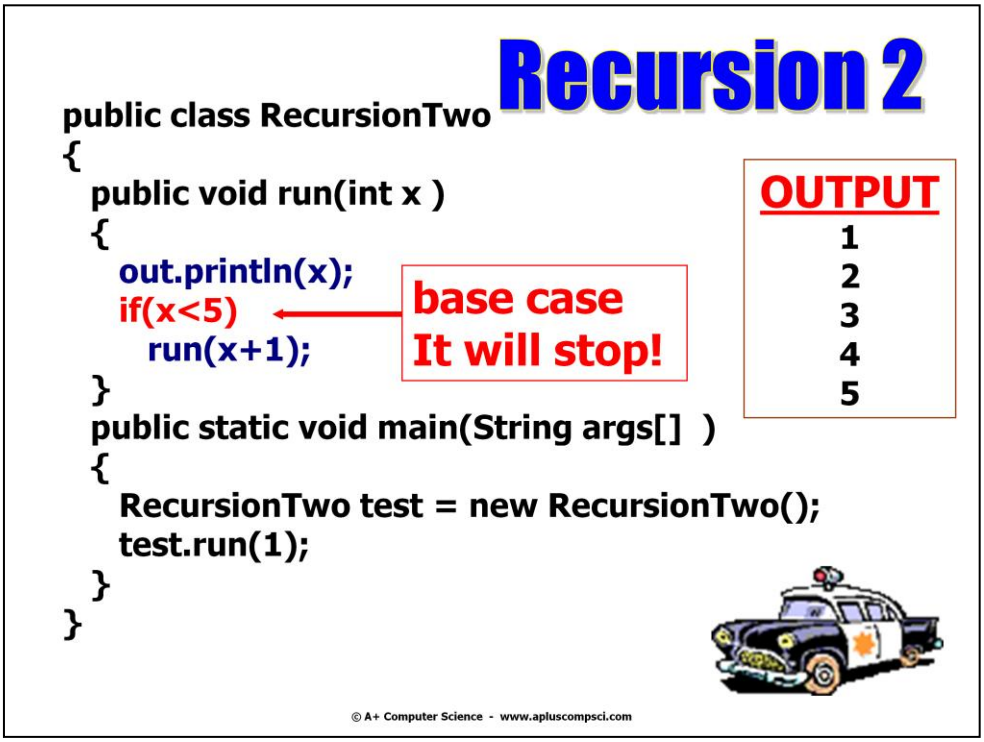
Once AR3 is finished, the execution sequence returns to AR2.



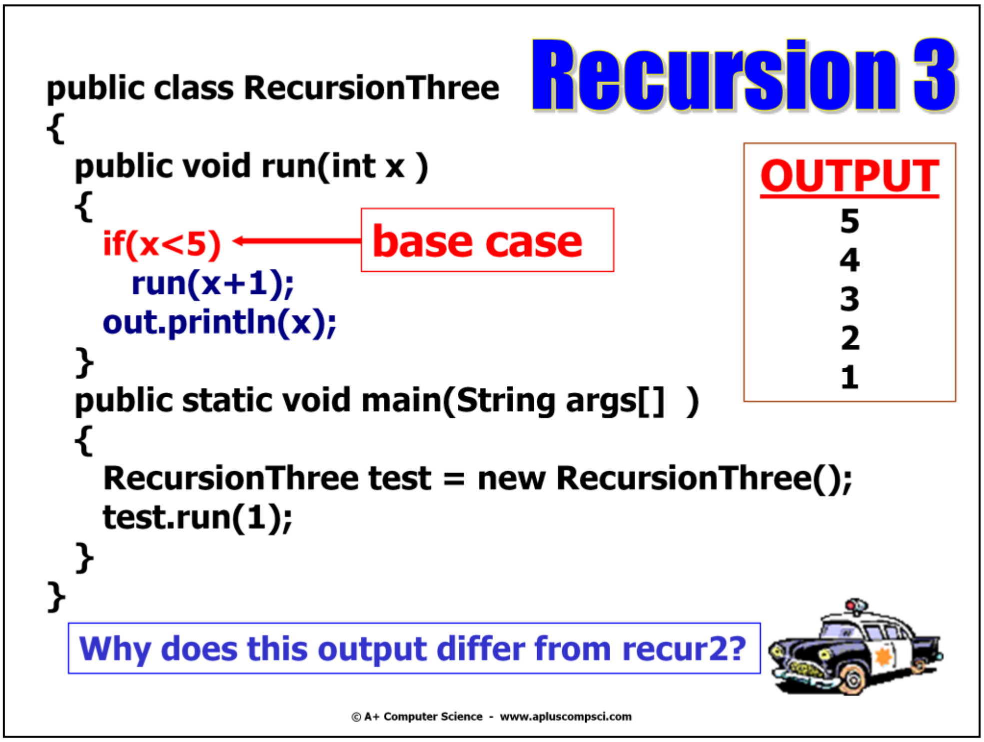
AR2 was placed on the stack after AR1and it is processed to completion after AR3 and before AR1.



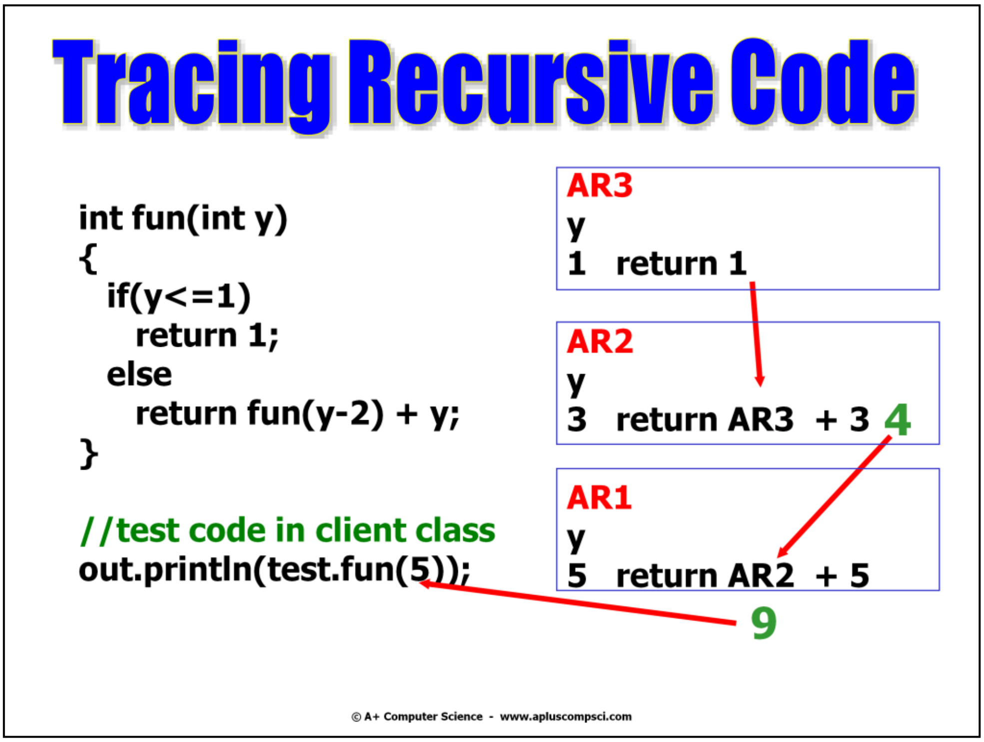
Because AR1 was placed on the stack first, it is processed to completion last.



Because the `println(x)` is before the recursive call, x is printed before the next AR is created on the stack.



Because the `println(x)` is after the recursive call, the `println(x)` is delayed until the new AR is completed. The `println(x)` will happen when the AR above it has finished.



As long as y is greater than 1, method fun() will continue to call itself creating recursion.

AR 1 - y = 5 return AR2 + 5

AR 2 - y = 3 return AR3 + 3

AR 3 - y = 1 return 1

Return 1 + 3 + 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));
```

AR3
x y
4 -1 return 4

AR2
x y
4 1 return AR3 + 4

8

AR1
x y
4 3 return AR2 + 4

12

open
recursionfour.java
recursionfive.java

© A+ Computer Science - www.apluscompsci.com

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 "";
}
```


In the example above, the recursive call occurs before a letter is appended.

The example method above will return a new String containing the same letters as `s` in the exact some order as s.

AR1 – `s="bat"   len=3   return` AR2 `+ t`

AR2 – `s="ba"   len=2   return` AR3 `+ a`

AR3 – `s="b"   len=1   return` AR4 `+ b`

AR4 – `s=""   len=0   return ""`

Final return `""+b+a+t`

split recursion
tail recursion

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


In the example above, the recursive call happens after a letter is appended.

The example method above will return a new String containing the same letters as `s` in the reverse order as s.

AR1 – `s="bat"  len=3  return t+`AR2

AR2 – `s="ba"  len=2  return a+`AR3

AR3 – `s="b"  len=1  return b+`AR4

AR4 – `s=""  len=0  return ""`

Final return `t+a+b+""`

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

AR2 – s="ab" return AR3 + b
AR1 – s="abc" return AR2 + c

The Stack

AR3 – s="a" return AR4 + a
AR2 – s="ab" return AR3 + b
AR1 – s="abc" return AR2 + c

The Stack

AR4 – s=""   return ""
AR3 – s="a"   return AR4 + a
AR2 – s="ab"   return AR3 + b
AR1 – s="abc"   return AR2 + c

The Stack

AR3 – s="a" return a
AR2 – s="ab" return AR3 + b
AR1 – s="abc" return AR2 + c

The Stack

AR2 – s="ab" return ab
AR1 – s="abc" return AR2 + c

© A+ Computer Science - www.apluscompsci.com

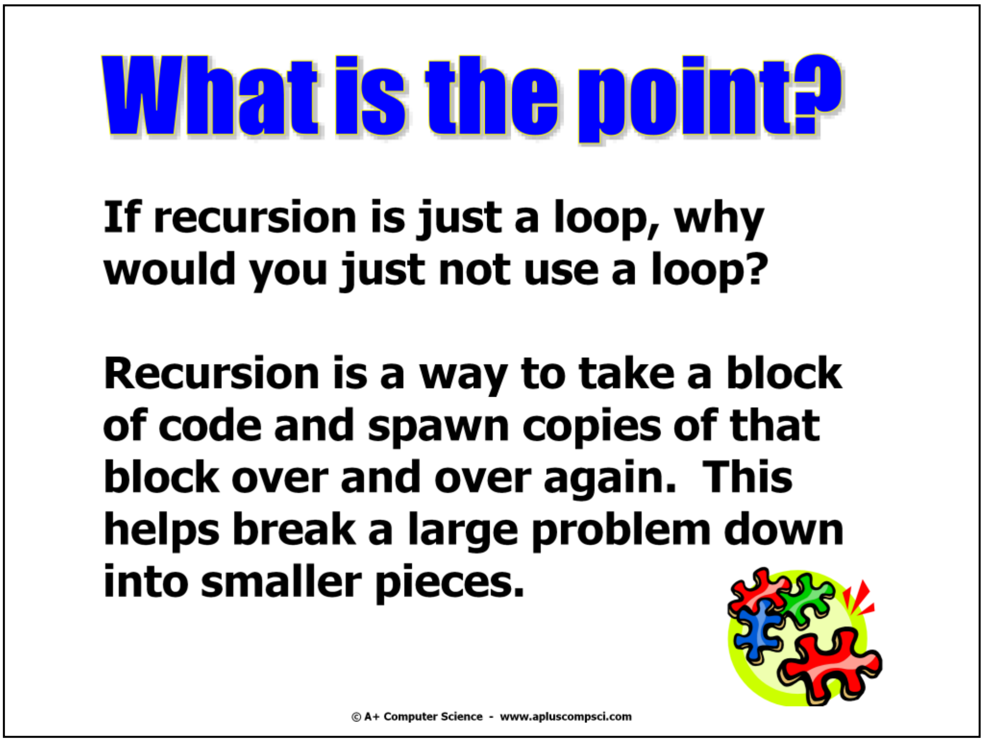
The Stack

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

OUTPUT

abc

AR1 – s="abc"
return abc



Using Recursion allows a section/block of code to be recreated while the program is running.

Each time the method is called, an instance of that method is created in memory. The size of the program can grow and shrink while the program is running.

A 10 line program might grow to a length of 1000 during run-time as recursive calls are made.

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.

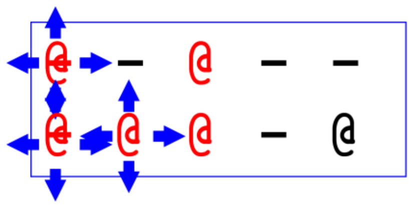
Each time this section of code is called, it checks around it for matching cells.

If a matching cell is found, a recursive call is made on that cell to check for its neighbors. This process continues as long as matching cells are found.

The original method is very short, but the actual code being used can get quite long during run time as the code grows dynamically.

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 21