COMP 110/L Lecture 15

Kyle Dewey

Outline

- Loops with arrays

Loops with Arrays

Loops with Arrays

Can *iterate* through arrays using loops

Loops with Arrays

Can *iterate* through arrays using loops

```
for (int x = 0; x < arr.length; x++) {
  System.out.println(x);
}
```

Loops with Arrays

Can *iterate* through arrays using loops

Not <=, since arrays start from 0

```
for (int x = 0; x < arr.length; x++) {
  System.out.println(x);
}
```

Example:
`PrintArgs.java`

Computing a Single Result

Common pattern: build a single result via iteration.
Update this result for each iteration.

Computing a Single Result

Common pattern: build a single result via iteration.
Update this result for each iteration.

Example: arithmetic product

-I'll start with specific examples, then move on to generalizing this

Computing a Single Result

Common pattern: build a single result via iteration.
Update this result for each iteration.

Example: arithmetic product

{ }

-First case: product of a list of no numbers

Computing a Single Result

Common pattern: build a single result via iteration.
Update this result for each iteration.

Example: arithmetic product

```
{ }
1
```

-First case: product of a list of no numbers
-This is defined to be 1

Computing a Single Result

Common pattern: build a single result via iteration.
Update this result for each iteration.

Example: arithmetic product
`{ }` `1`
`{5}`

-Second case: product of a list containing a single number

Computing a Single Result

Common pattern: build a single result via iteration.
Update this result for each iteration.

Example: arithmetic product

```
{}
1
```

```
{5}
1 * 5
```

-Second case: product of a list containing a single number
-This is defined as 1 * that number, always yielding that number

Computing a Single Result

Common pattern: build a single result via iteration.
Update this result for each iteration.

Example: arithmetic product

```
{}
1
```

```
{5}
1 * 5
5
```

-Second case: product of a list containing a single number
-This is defined as 1 * that number, always yielding that number

Example: arithmetic product

Example: arithmetic product

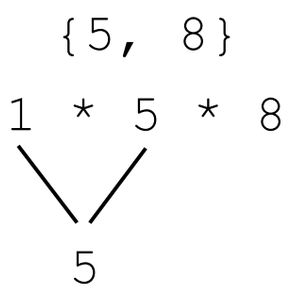
```
{5, 8}
```

Example: arithmetic product

```
{5, 8}
```

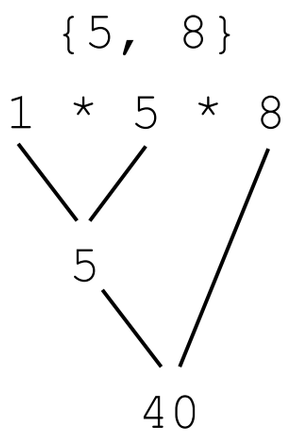
```
1 * 5 * 8
```

Example: arithmetic product



-Instead of doing all the multiplications at once, I'll do them stepwise, which mirrors what the code will need to do

Example: arithmetic product



-Instead of doing all the multiplications at once, I'll do them stepwise, which mirrors what the code will need to do

Example: arithmetic product

Example: arithmetic product

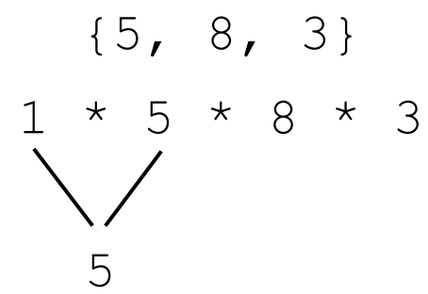
```
{5, 8, 3}
```

Example: arithmetic product

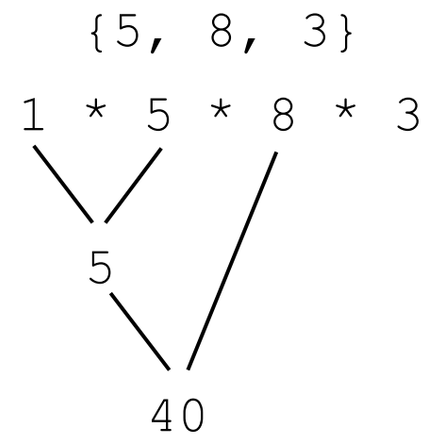
```
{5, 8, 3}
```

```
1 * 5 * 8 * 3
```

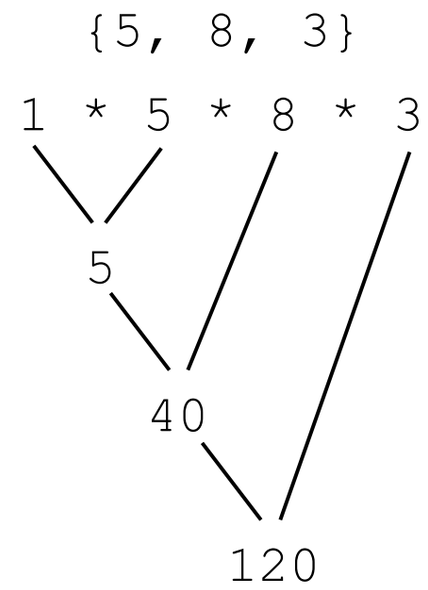
Example: arithmetic product



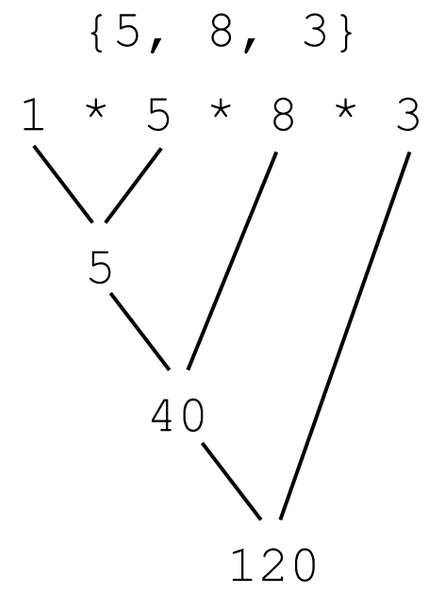
Example: arithmetic product



Example: arithmetic product

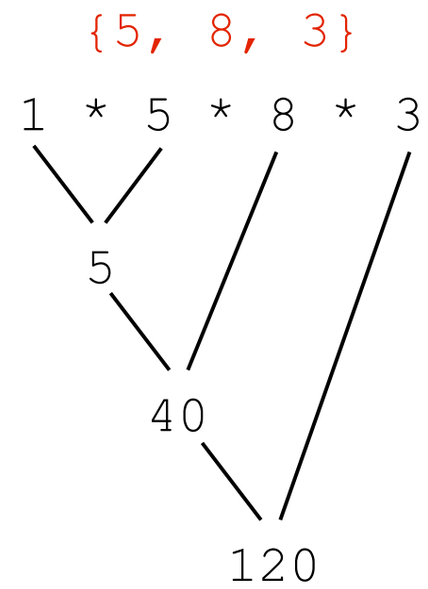


In Code



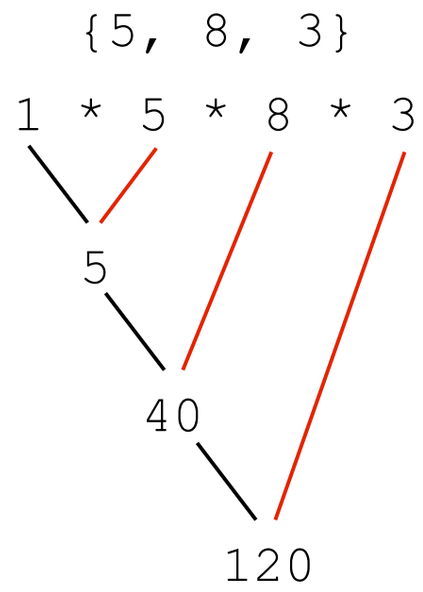
Variables needed:

In Code



Variables needed: array

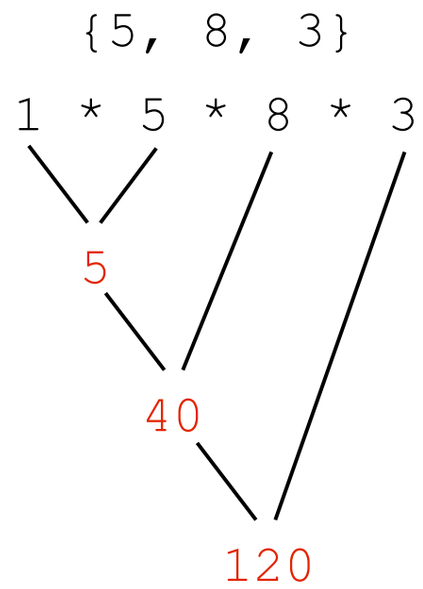
In Code



Variables needed: array, position in array

-Position in the array changes over time

In Code



Variables needed: array, position in array, result

-Like the position, the result changes over time
-Only once we've completed going through the whole array is the result final

Example

- `Product.java`
- `ProductTest.java`

Another example: arithmetic sum

Another example: arithmetic sum

{ }

Another example: arithmetic sum

{ }

0

Another example: arithmetic sum

{ }

0

{ 2 }

Another example: arithmetic sum

$\{\}$

0

$\{2\}$

$0 + 2$

Another example: arithmetic sum

{ }

0

{ 2 }

0 + 2

2

Another example: arithmetic sum

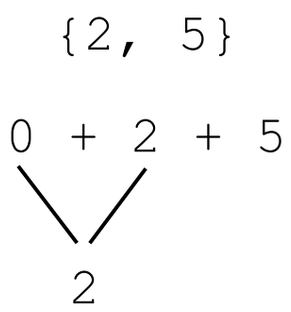
```
{2, 5}
```

Another example: arithmetic sum

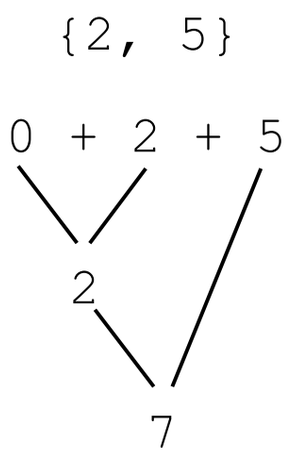
```
{2, 5}
```

```
0 + 2 + 5
```

Another example: arithmetic sum



Another example: arithmetic sum



Another example: arithmetic sum

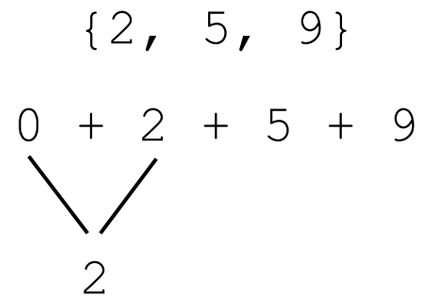
```
{2, 5, 9}
```

Another example: arithmetic sum

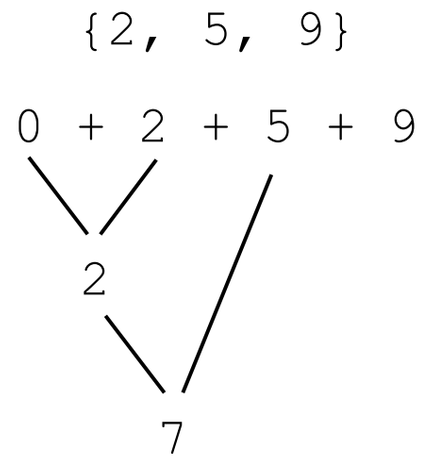
```
{2, 5, 9}
```

```
0 + 2 + 5 + 9
```

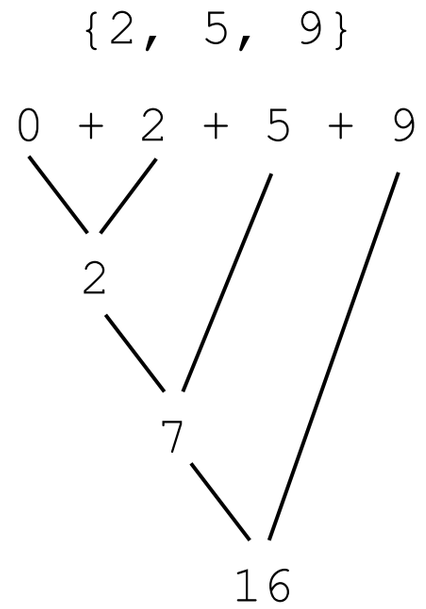
Another example: arithmetic sum



Another example: arithmetic sum



Another example: arithmetic sum



General Pattern

General Pattern

```
ResultType result = initialResult;
```

-Initialize the result to some initial value
-For product, this is 1
-For sum, this is 0

General Pattern

```
ResultType result = initialResult;
for (int index = whereToStart;
```

-Start iterating through the array, starting from some starting position
-For product and sum, this starting position is 0 (the first index of the array)

General Pattern

```
ResultType result = initialResult;
for (int index = whereToStart;
      index < whereToEnd;
```

-Continue iterating until some stopping condition
-For both product and sum, this should be array.length

General Pattern

```
ResultType result = initialResult;
for (int index = whereToStart;
     index < whereToEnd;
     index++) {
```

-Keep doing this iteration for every index of the array

General Pattern

```
ResultType result = initialResult;
for (int index = whereToStart;
     index < whereToEnd;
     index++) {
  result = oneStep(array[index],
                   result);
}
```

-For each iteration, perform some computation involving the current element of the array (determined by both the array and whatever array index we are on), along with the current result
-For product, this is result *= array[index] (AKA result = result * array[index])