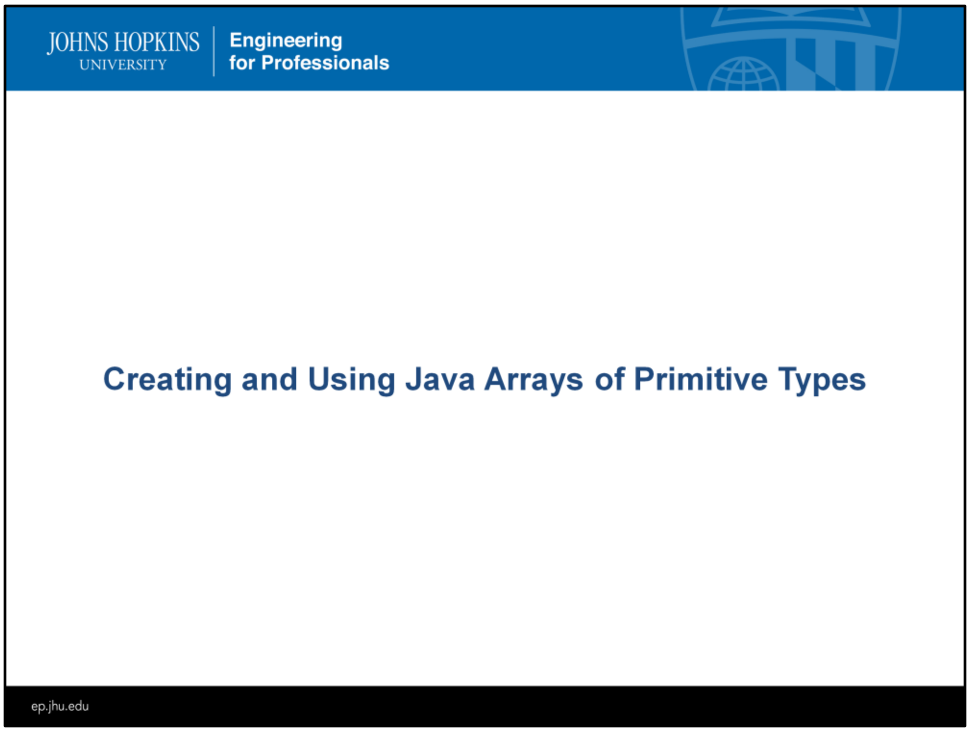
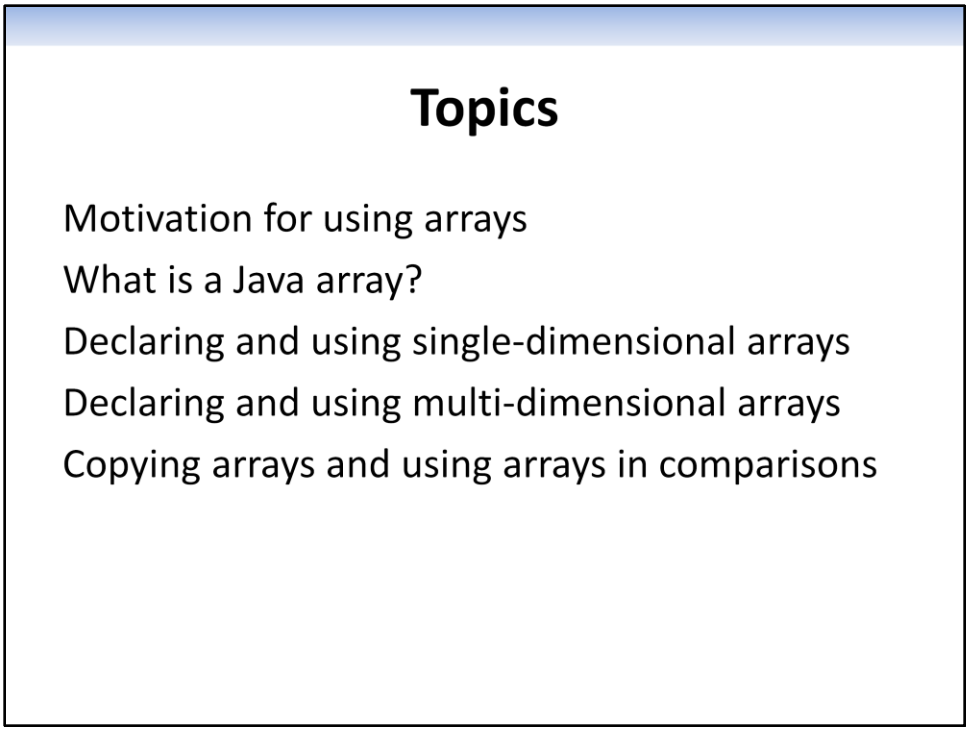

JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

Creating and Using Java Arrays of Primitive Types

ep.jhu.edu

This presentation is a brief discussion on creating and using Java arrays of primitive types. Java primitive types are the built-in types for example int, float, char, boolean, etc. Discussions on using arrays with objects are in a separate presentation.

Java arrays are very useful in solving many programming problems. So lets see how we can create and use Java arrays.



Topics

Motivation for using arrays

What is a Java array?

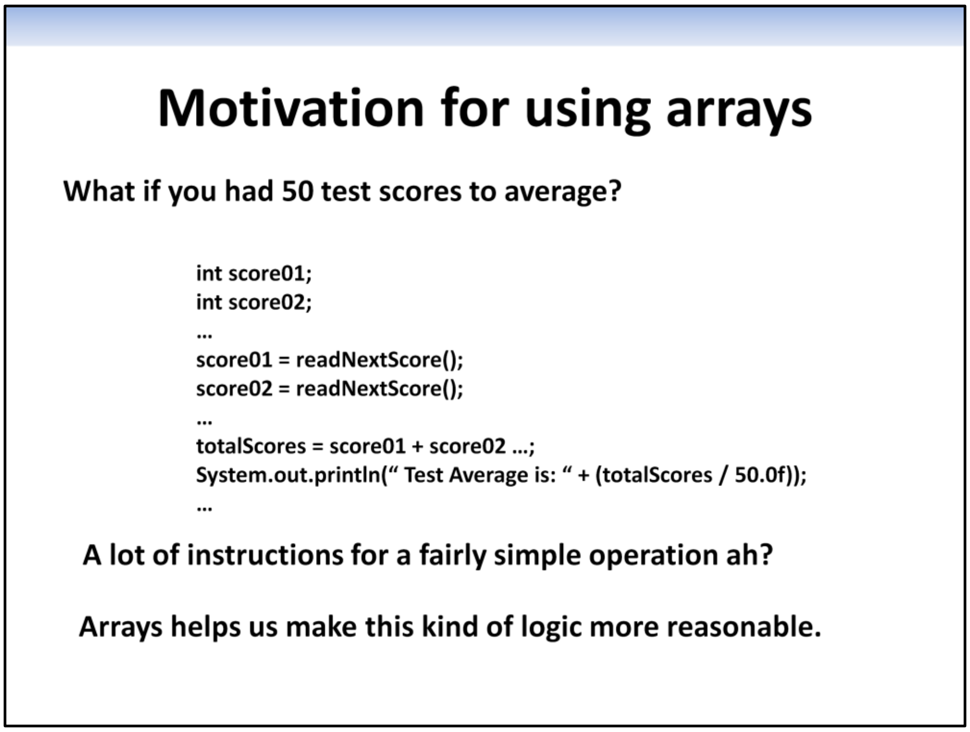
Declaring and using single-dimensional arrays

Declaring and using multi-dimensional arrays

Copying arrays and using arrays in comparisons

The main topic s of this presentation are motivation for using arrays, what is a Java array, declaring and using single-dimensional arrays, and declaring and using multi-dimensional arrays. We will finish off with discussing copying arrays and using arrays in comparisons.

Some interesting stuff so lets get to it!



Motivation for using arrays

What if you had 50 test scores to average?

```
int score01;
int score02;
…
score01 = readNextScore();
score02 = readNextScore();
…
totalScores = score01 + score02 …;
System.out.println(" Test Average is: " + (totalScores / 50.0f));
…
```

A lot of instructions for a fairly simple operation ah?

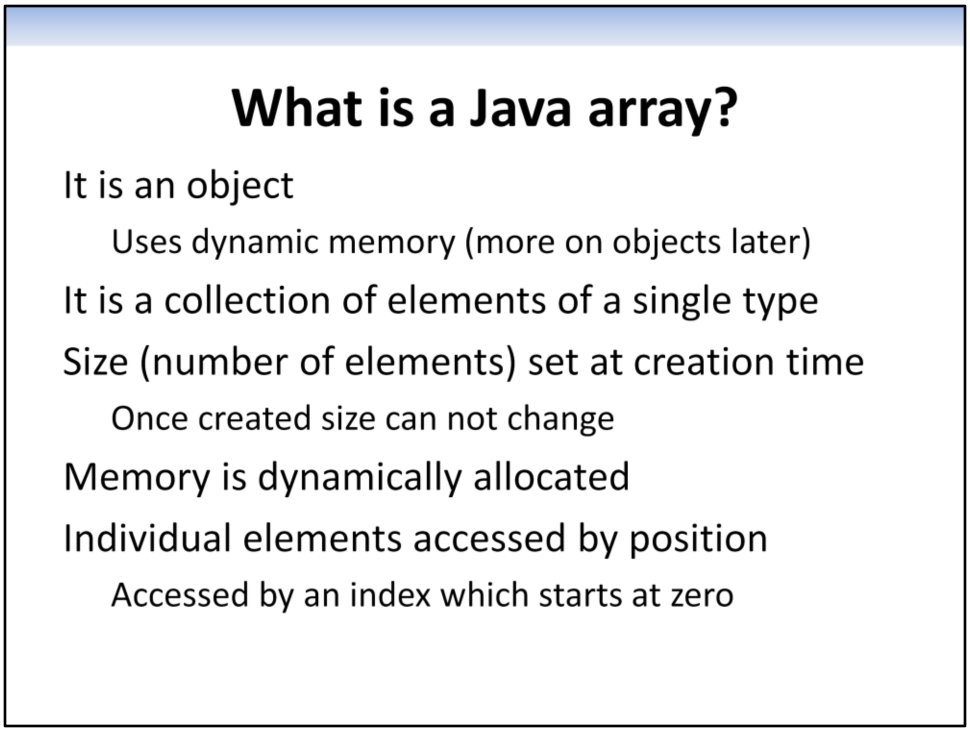
Arrays helps us make this kind of logic more reasonable.

Lets look at one main motivation for using Java arrays.

Lets say you have 50 test scores you have to average in your Java program. How are you going to store these test scores? Well you could declare 50 variables then assign a single test score to each one, for example maybe reading them from a file, then do a calculation to average the scores.

This is a lot of instructions to do a simple average of 50 numbers. What if we had 100 scores, a 1,000 scores? This is a lot of work.

Arrays make this type of task much more reasonable to specify in your code. Lets see how.



What is a Java array?

It is an object
- Uses dynamic memory (more on objects later)

It is a collection of elements of a single type

Size (number of elements) set at creation time
- Once created size can not change

Memory is dynamically allocated

Individual elements accessed by position
- Accessed by an index which starts at zero

So what is a Java array?

It is an object. Java objects are more complex and versatile than Java primitive types. Discussions on objects in general are left to separate presentations but array are objects. In this presentation we will stick to arrays which use Java built-in primitive types.

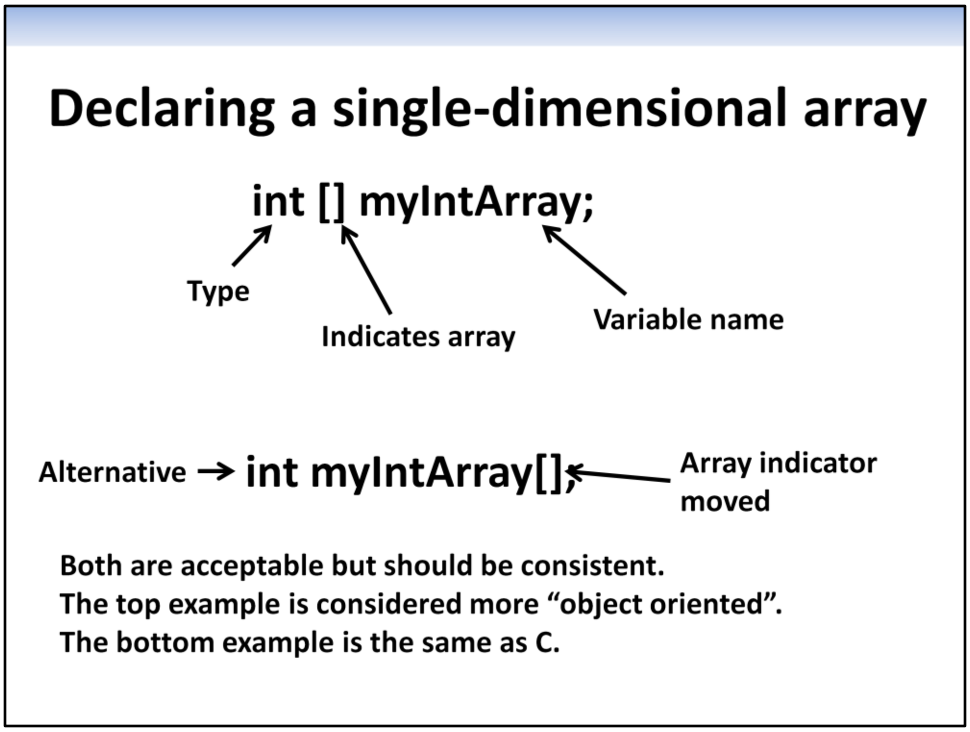
An array is a collection of elements of a single type. For example you could have an array of integers or an array of characters. For a single array, all elements must be of the same type. No mix-and-matching of types here. Each element however, can and usually do have different values.

The number of elements in an array is specified when the array is created. Once created, the size of the array (and the amount of memory it occupies) is fixed.

Memory for an array is dynamically allocated when created. Just declaring an array does not allocate memory.

Individual array elements are access by index position. Indexes always start at zero.

Quite a lot here so lets look at some details.



Declaring a single-dimensional array

```
int [] myIntArray;
```

Type
Indicates array
Variable name

Alternative → `int myIntArray[];` ← Array indicator moved

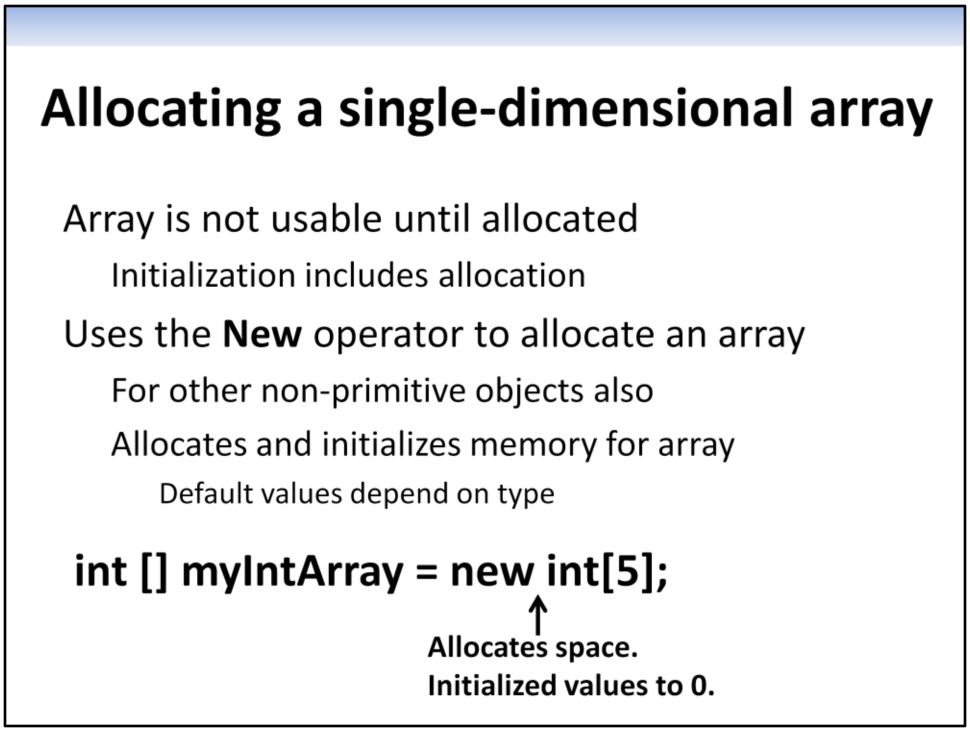
Both are acceptable but should be consistent.
The top example is considered more "object oriented".
The bottom example is the same as C.

Lets look at how to declare a single-dimensional array..

Here we have a declaration of a single-dimensional Java array. We start with the type of elements the array is going to hold. Then the closed brackets indicate this is an array but the size is not specified yet. Then comes the variable name given to the array. So except for the closed brackets, the declaration of a single-dimensional array is the same as a primitive-type variable.

Java allows an alternative way of specifying arrays as shown here. The only difference is moving the closed brackets to after the variable name instead of after the type.

Both syntaxes are acceptable but you should pick one and stick with it. Most newer Java programs tend to use the top syntax since it is considered more "Object Oriented". The bottom syntax is the way arrays are declared in C and can be done this way in C++ also. It helps programmers of these languages to have an easier time programming with Java.



Allocating a single-dimensional array

- Array is not usable until allocated
  - Initialization includes allocation
- Uses the **New** operator to allocate an array
  - For other non-primitive objects also
  - Allocates and initializes memory for array
    - Default values depend on type

int [] myIntArray = new int[5];

$\uparrow$

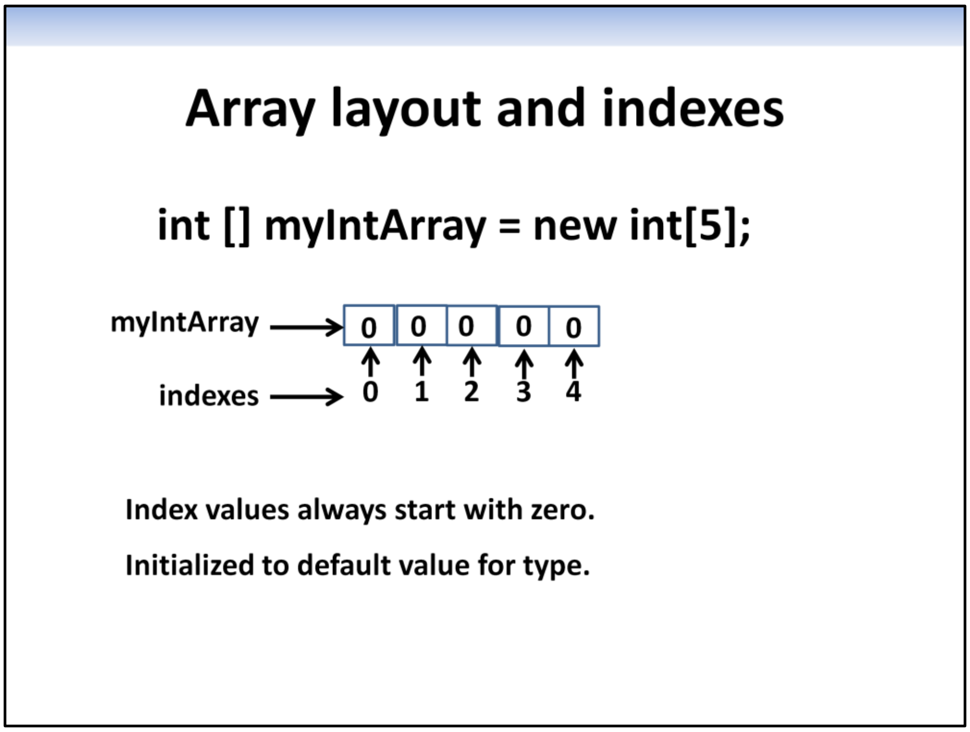
Allocates space.
Initialized values to 0.

Remember declaring an array does not make it usable. Memory must be allocated for the array and Java needs to know the size of the array for it to be usable. You can perform all of these actions in a single statement with initialization but this topic will be covered soon. For now lets see how to allocate a single-dimensional array.

You use the new operator to allocate an array. The new operator is used by all Java objects but we will concentrate on arrays here. The new operator allocates memory for the array as calculated using the specified size and type. Once the memory is allocated for the array, the Java Virtual Machine (JVM) initializes the array elements to their default values. The default value depends on the type of array. For example the default value for a int is zero.

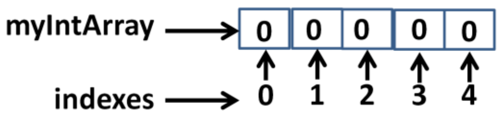
Here is an example of allocating an array. Note the new operator is specified, then the type of array (int in this case), then the number of elements (AKA size) is specified inside of brackets. This statement will allocate space for five integers and initialize them to zero.

Lets take a look at this array in a bit more detail.



Array layout and indexes

int [] myIntArray = new int[5];



Index values always start with zero.

Initialized to default value for type.

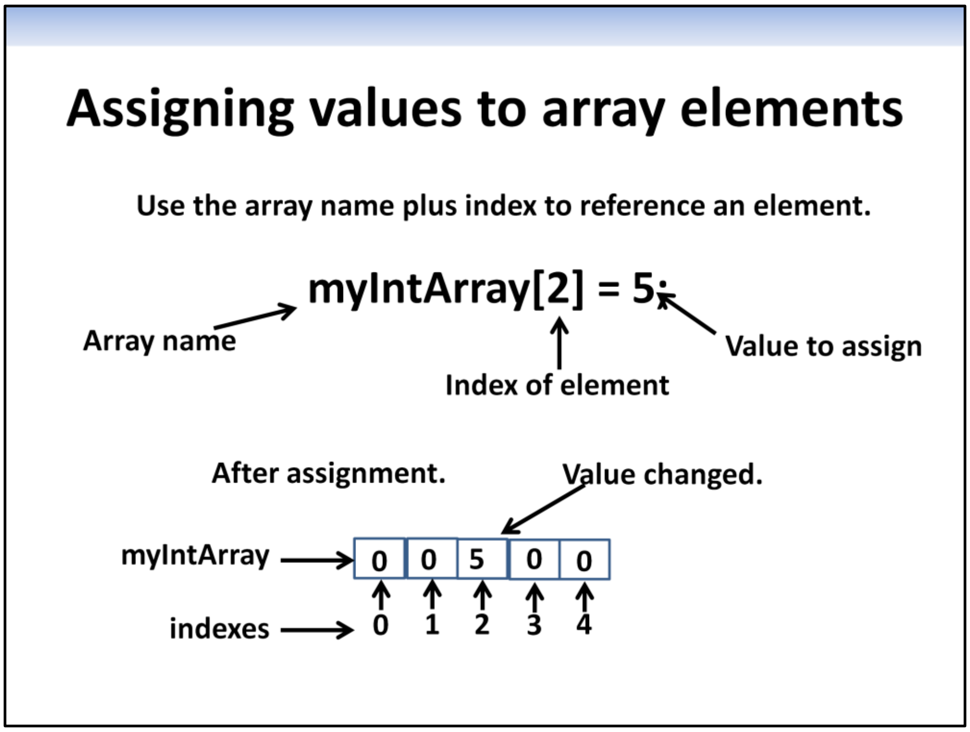
Here is the array allocation we looked at just a moment ago. It is going to allocate five integer locations and initialize their values to zero. Letts look at this a bit more graphically.

Here we see myIntArray is a reference (like a C/C++ pointer but different) to a 5-element integer array.

Note the indexes start at zero then increment by one.

The array elements are initialized to the default value for the specified type which is zero in this case.

We usually don't want array elements to remain in their initialized state for the entire run of the program so lets see how to assign a value to a single array element.



Assigning values to array elements

Use the array name plus index to reference an element.

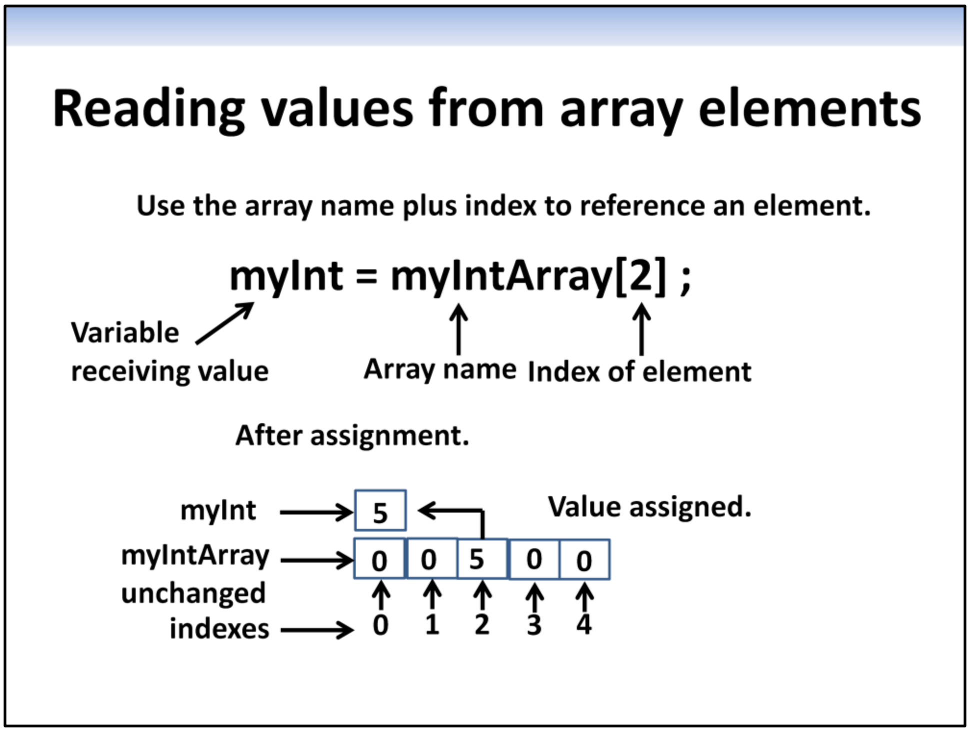
To assign a value to a single array element, you need to specify the array name, the index of the element in the array you want to modify, and the value you want the array element to be assigned.

Here is an example which assigns the integer value 5 to the third element (index value 2) of the myIntArray. You need to specify the array name, the index of the element you want to modify in brackets, then the value you want assigned to the element..

Note it is the third element but its index is two since indexes start at zero. Note for the rest of this presentation I will refer to the element by its index value with indexes starting a zero instead of their position relative to starting at one.

After the assignment the myIntArray looks like this. Notice the element with the index of two has been modified.

Now since we have assigning a value to an array element, lets look at how to read the value of an array element.



Reading values from array elements

Use the array name plus index to reference an element.

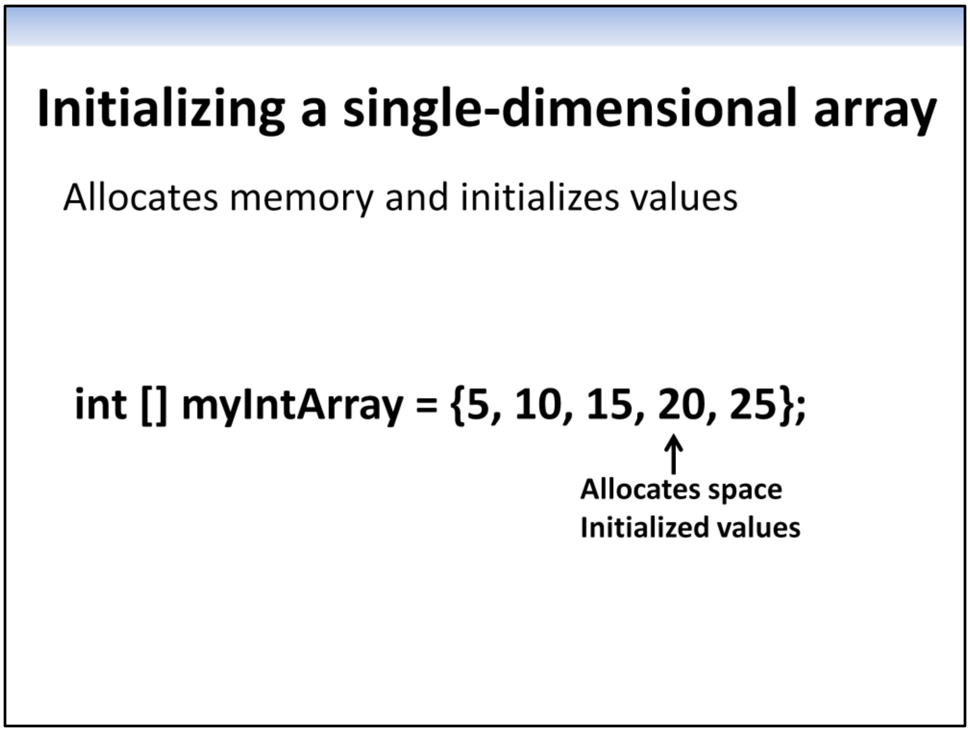
```
myInt = myIntArray[2] ;
```

Reading an array element value is very similar to modifying an array element value except you swap positions relative to the assignment operator. Here you need to specify the array name, the array element you want to read the value of by using its index, and where you want the read value to go.

Here is an example of reading the second value of the myIntArray and assigning the value to the variable myInt. Here we have the variable, myInt, where the read value is going to go, the name of the array, and the index of the element we want the value of. Except for the bracketed index value, it is like any assignment statement.

After the assignment the myIntArray array and myInt values look like this. Note the value of the second element of the myIntArray has been copied to the myInt variable. Also note the myIntArray is unchanged.

If we know what values we want in an array when we declare it we can initialize the array at declaration time. Lets see how to do that.



Initializing a single-dimensional array

Allocates memory and initializes values

```
int [] myIntArray = {5, 10, 15, 20, 25};
```

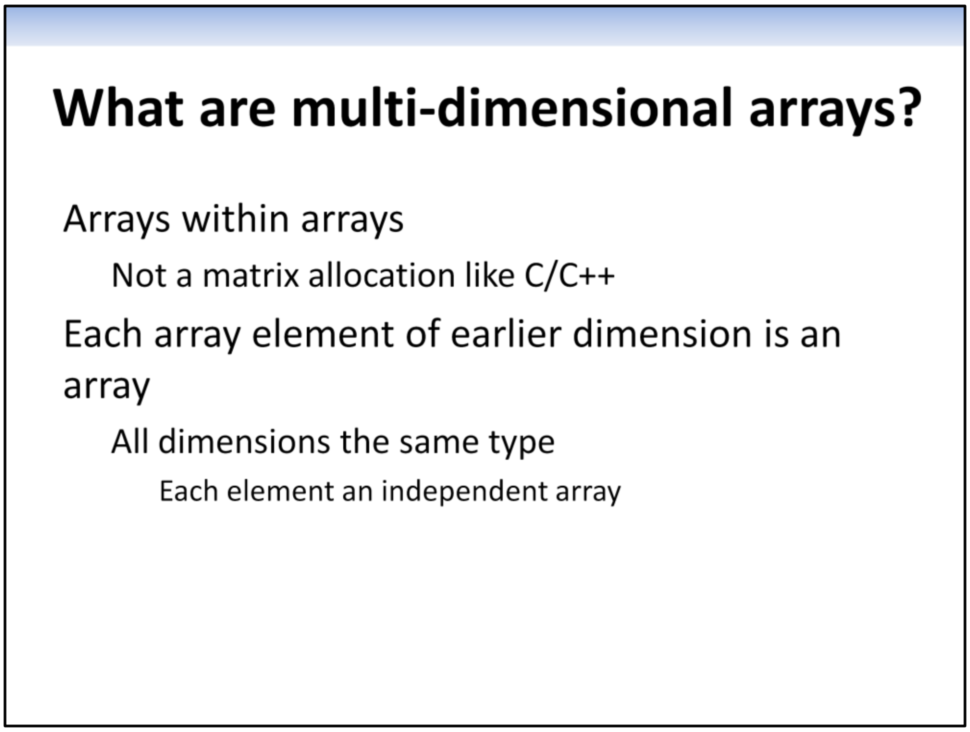
$\uparrow$

Allocates space
Initialized values

You can initialize an array the same time you declare it.   This technique declares the array, allocate space for it, and loads the specified values into the array elements.

Here is an example of a single-dimensional array being declared, allocated, and initialized in one statement. This statement will create a 5-element integer array and load the values specified in the 5 elements with 5 in element zero, 10 in element one, etc.

Now lets take a look at multi-dimensional arrays.



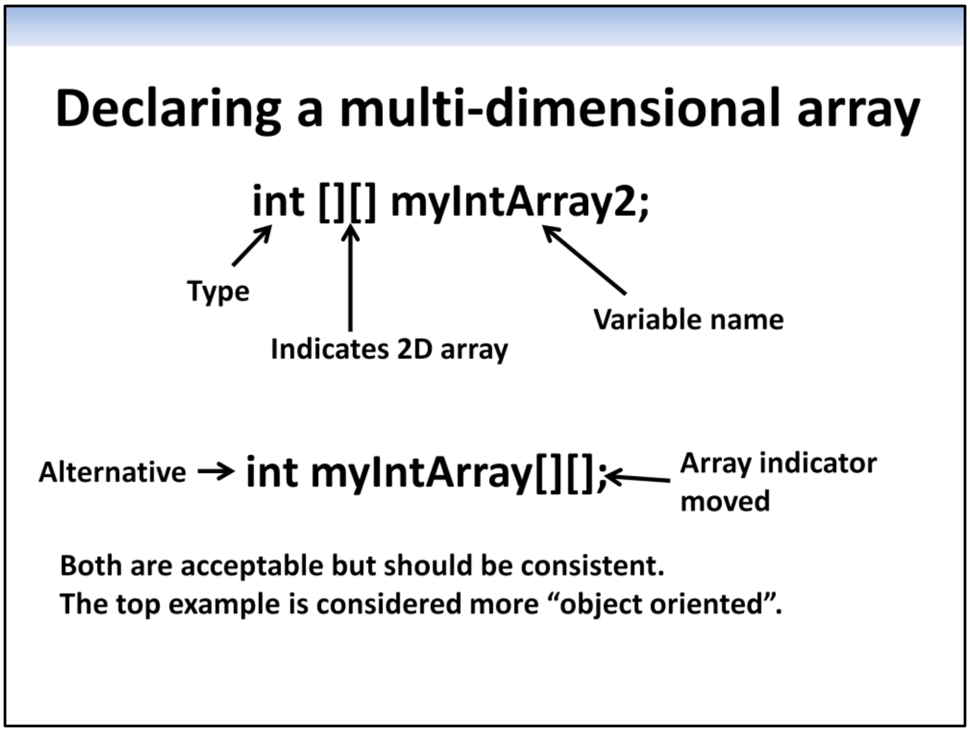
What are multi-dimensional arrays?

- Arrays within arrays
  - Not a matrix allocation like C/C++
- Each array element of earlier dimension is an array
  - All dimensions the same type
    - Each element an independent array

In Java, multidimensional arrays are arrays inside of arrays. This is different than the matrix memory organization of multi-dimensional arrays in C and C++ although you can simulate this arrangement with Java arrays.

In Java, each array element of a multi-dimensional array except the last dimension is an array, not an individual element. All the elements of the last dimension are of the same type. All elements of dimensions other than the last are arrays of that type.

This can get confusing so for now we will sick with looking at 2-dimensional arrays. Lets take a look at declaring a 2-dimensional array.



Declaring a multi-dimensional array

```
int [][] myIntArray2;
```

Type
Indicates 2D array
Variable name
Alternative →
Array indicator moved

```
int myIntArray[][];
```

Both are acceptable but should be consistent.
The top example is considered more “object oriented”.

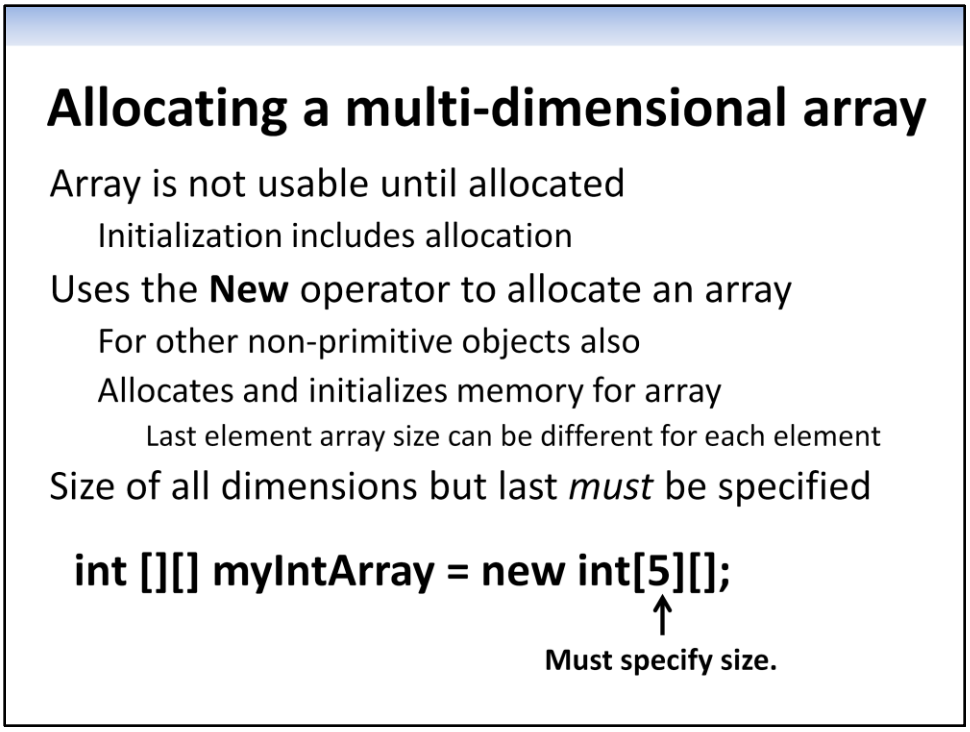
Declaring a multi-dimensional array is similar to declaring a single-dimensional array. The only change is you need you supply a bracket-pair for each dimension you want.

Here is an example of declaring a 2-dimensional array of integers. Like with single-dimensional array declarations, you start with the type of the array, then you indicate the number of dimensions with the number of bracket-pairs listed, then what name the array will have.

Just like with single-dimensional array declaration, you can use the alternative syntax with the array dimensions specified after the variable name.

Both syntaxes are acceptable but as mentioned before you should be consistent. The top example is considered more “Object Oriented” while the bottom example is the same as syntax used in C and C++.

Now since we now know how to declare a multi-dimensional array, lets see how to allocate one.



Allocating a multi-dimensional array

Array is not usable until allocated
- Initialization includes allocation

Uses the **New** operator to allocate an array
- For other non-primitive objects also
- Allocates and initializes memory for array
  - Last element array size can be different for each element

Size of all dimensions but last *must* be specified

```
int [][] myIntArray = new int[5][];
```

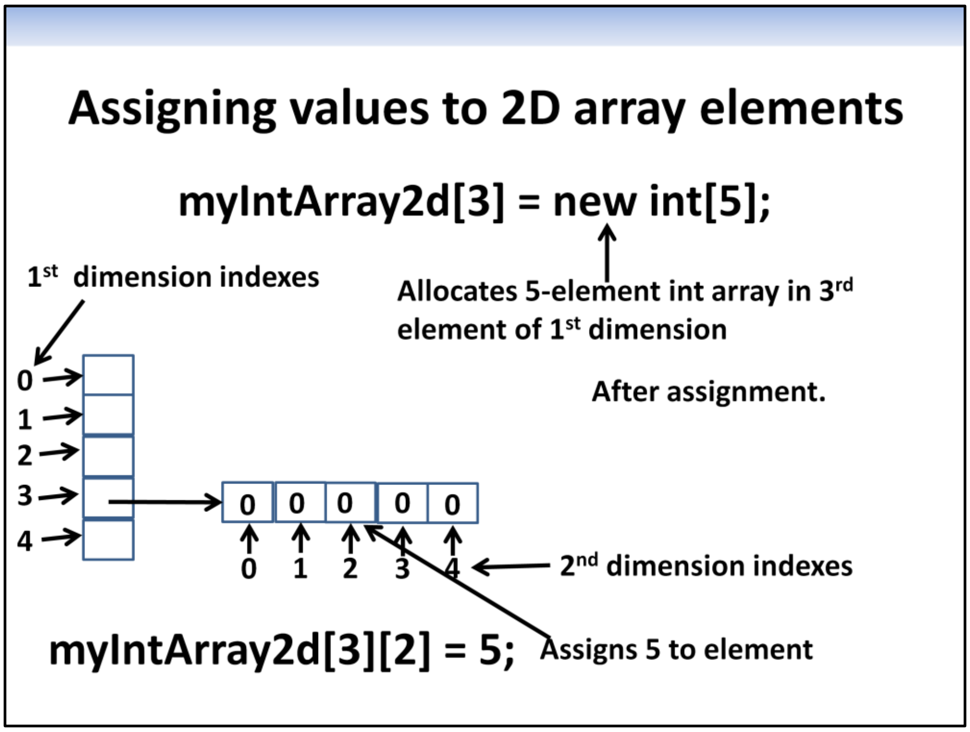
Allocating multi-dimensional arrays is similar to allocating single-dimensional arrays but there are some differences.

Just like with single-dimensional arrays, the array is not usable until it is allocated. To allocate a multi-dimensional array you use the new operator. As mentioned previously, the new operator is used to allocate other objects in addition to arrays. Just like in single-dimensional arrays, allocation initializes element values to their default type. Here, however, usually the element being initialize is an array of a type not an element of a type. The default type of an array is a null reference.

The size of all dimensions but the last one must be specified. This is actually the same as with single-dimensional arrays since there is only one dimension specified that one dimension is the last one and the number of elements must be specified.

Here is an example of an allocation of a 2-dimensional array. Note the size of all dimensions are specified except the last one.

Lets take a look at how to assign values to a 2-dimensional array.



Assigning values to 2D array elements

```
myIntArray2d[3] = new int[5];
```

```
myIntArray2d[3][2] = 5;
```

Assigning values to a 2-dimensional array is a bit more complex than a single-dimensional array. Remember for a 2-dimensional array the first dimension contains arrays of the specified type, not elements of the specified type.

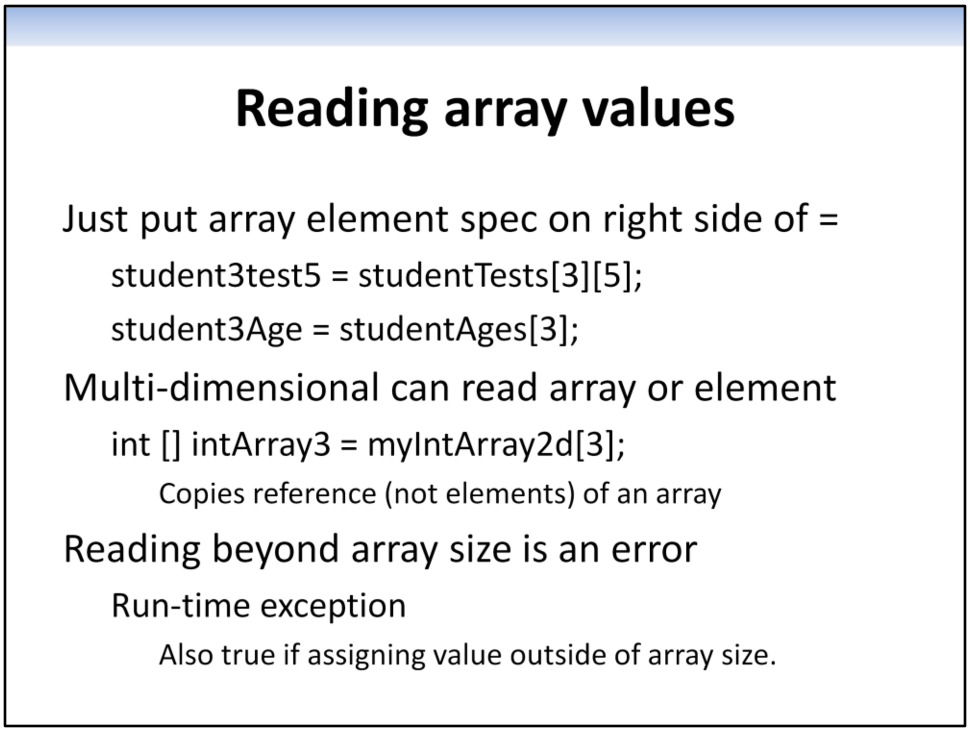
Her is an example allocating an array of integers to the 3rd element of the first dimension of the myIntArray2d. Note we are allocating a five-element integer array to this element, not an integer.

After the assignment the myIntArray2d array looks like this. Notice the 5-element integer array is referenced by the 3rd element of the 1st dimension. Also note the other 1st dimension elements are empty (null). Also notice the 5-element integer array's elements have been initialized to zero.

Here is an example of assigning an integer value of 5 to the 2nd element of the array referenced by the 3rd element of the 1st dimension of the 2-dimensional array.

You should pause this presentation until you understand what is going on here. It is a bit complicated so take your time.

Now lets move on to reading values from a 2-dimensional array.



Reading array values

Just put array element spec on right side of =

- student3test5 = studentTests[3][5];
- student3Age = studentAges[3];

Multi-dimensional can read array or element

- int [] intArray3 = myIntArray2d[3];
  - Copies reference (not elements) of an array

Reading beyond array size is an error

- Run-time exception
  - Also true if assigning value outside of array size.

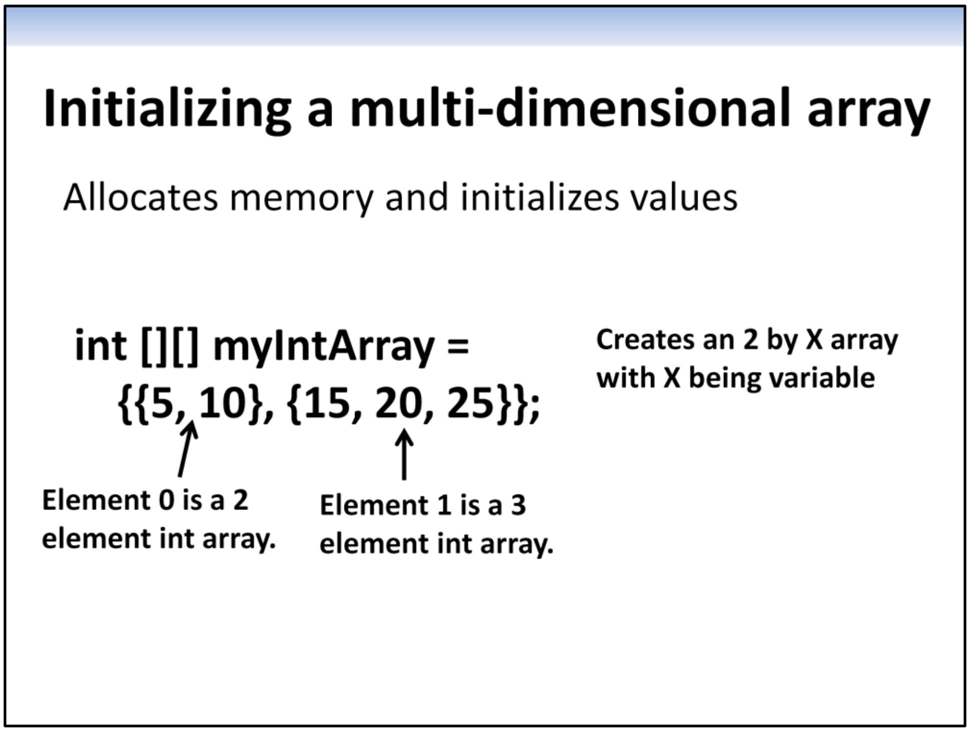
Reading multiple -dimensional array values is similar to reading single-dimensional array values. Just the more dimensions you have the more indexes you may need to specify. Easier said than done right?

To read array values, you basically put the array element specification on the right side of the assignment operator with the variable to receive the array element's value on the left side of the assignment operator as shown in these examples.

Note when reading a multi-dimensional array by less than the last dimension, you are copying the reference, not the data of an array. We will talk about this in just a bit.

In all cases, reading beyond the array size or using an index less than zero will result in a run-time error (AKA an exception). This is true if reading or writing to an array.

Lets move on and look at initializing a multi-dimensional array.



Initializing a multi-dimensional array

Allocates memory and initializes values

```
int [][] myIntArray =
   {{5, 10}, {15, 20, 25}};
```

Element 0 is a 2 element int array.

Element 1 is a 3 element int array.

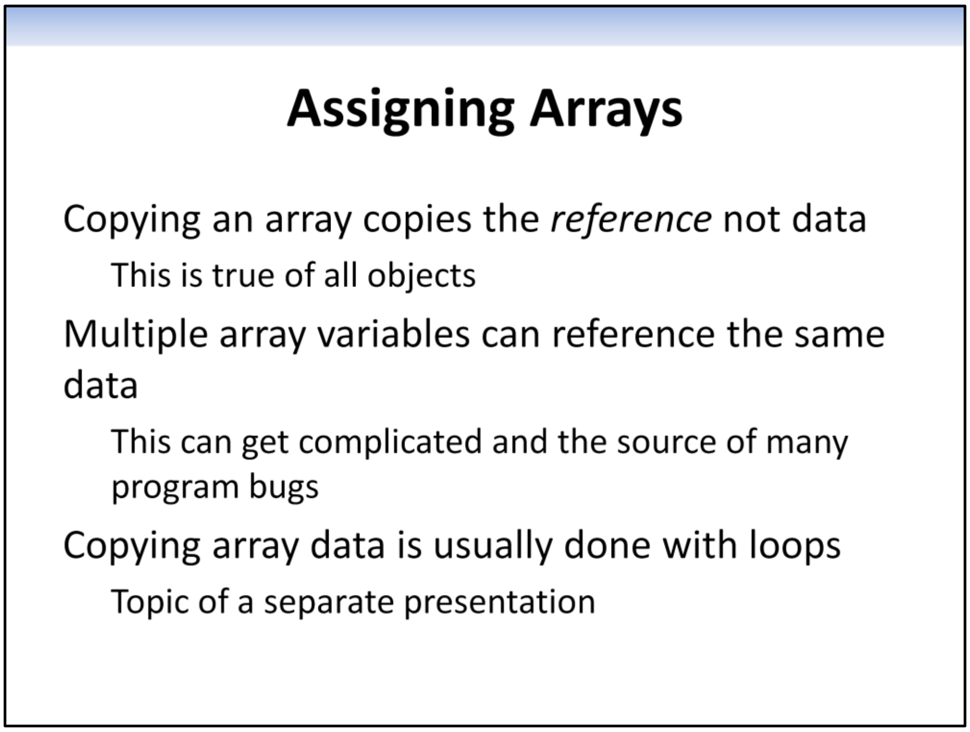
Creates an 2 by X array with X being variable

Initializing a multi-dimensional array is similar to initializing a single-dimensional array but with a bit of added complexity.

As with a single-dimensional array, initializing an multi-dimensional array at declaration time declares the array, allocates memory for it, and initializes the elements to the specified values.

Here we have an example of declaring, allocating, and initializing a 2-dimensional integer array. The initialization is similar to a single-dimensional array except values of each 2nd dimension array are specified in their own set of braces with a comma between each set of values. Note the sizes of the 2nd dimensional array are different with the zero element array containing an array of two integers while the 1 element array containing an array of three integers.

This ability to have different array sizes on the last dimension is a flexibility standard C and C++ arrays do not have.



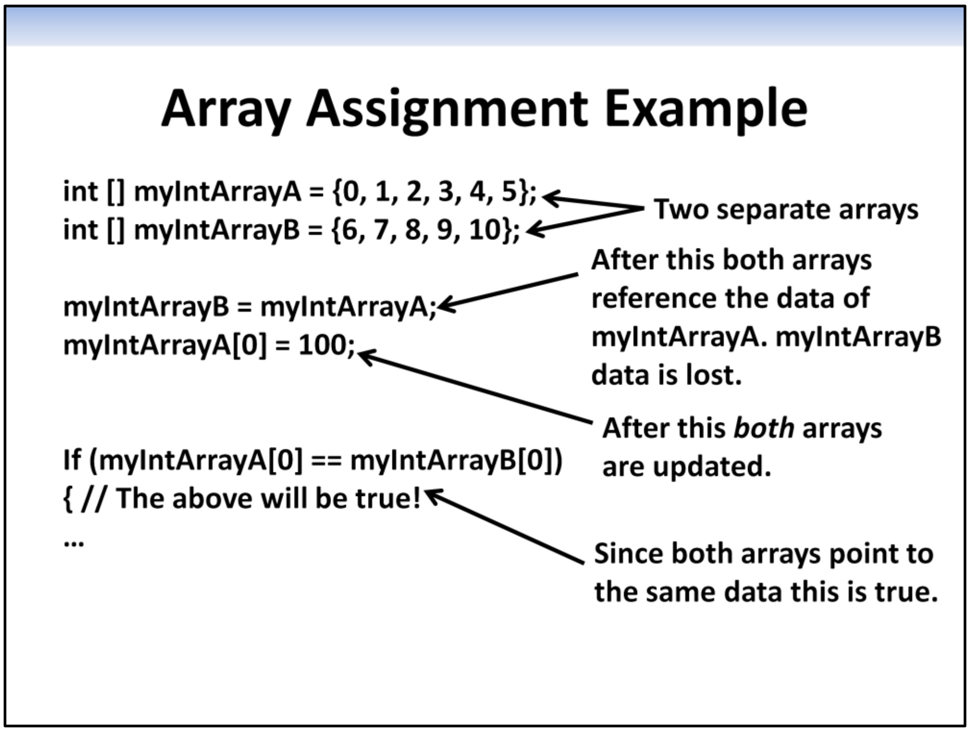
Assigning Arrays

- Copying an array copies the *reference* not data
  - This is true of all objects
- Multiple array variables can reference the same data
  - This can get complicated and the source of many program bugs
- Copying array data is usually done with loops
  - Topic of a separate presentation

Assigning Java arrays to array variables is not like assigning primitive types to primitive variables. When assigning arrays, you are copying the *reference* not the data of the array.

The assignment of array references results in after an assignment of an array from another array, both array variables reference the same data. This has some interesting characteristics as we will see soon. It is easy to lose track of what array variable is pointing to what data and as a result this issue is the source of many Java programming bugs.

If you want to copy the data elements of one array to another, this is usually done with loops which is the subject of a separate presentation.



Array Assignment Example

```
int [] myIntArrayA = {0, 1, 2, 3, 4, 5};
int [] myIntArrayB = {6, 7, 8, 9, 10};

myIntArrayB = myIntArrayA;
myIntArrayA[0] = 100;

If (myIntArrayA[0] == myIntArrayB[0])
{ // The above will be true!
…
```

Two separate arrays

After this both arrays reference the data of myIntArrayA. myIntArrayB data is lost.

After this *both* arrays are updated.

Since both arrays point to the same data this is true.

Here is an example of assigning one array to another and how it could get confusing.

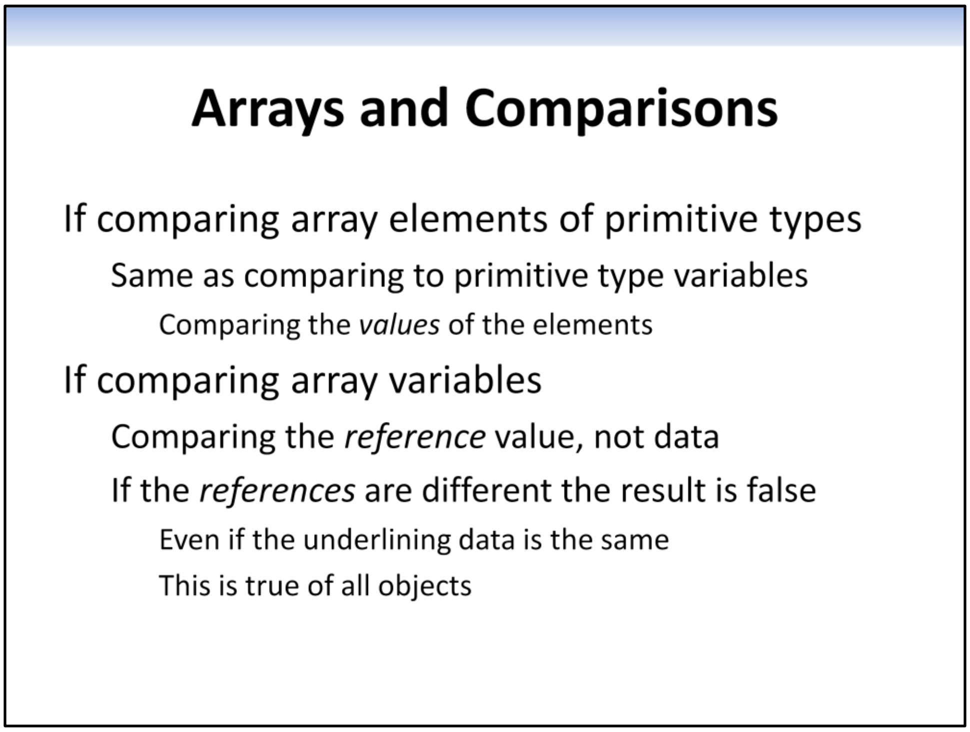
We start of by defining two separate integer arrays, myIntArrrayA and myIntArrayB each initialized with a different set of values.

Next we copy myIntArrayA to myIntArrayB but wait! We are coping the myIntArrayA *reference* not the data. Lets see how this plays out.

Next we assign the integer value 100 to the zero element of myIntArrayA. Because they both now contain a reference to the myIntArrayA array, it updates *both* arrays.

Now we compare the zero element array value of the two arrays. Again, because they both reference the same underlining array data, the comparison returns a value of true. Unless you concentrated on the code and kept track of the array references, this result may not have been what you expected.

In the last part of this example an if statement was used to compare two array elements. Comparing arrays is our next topic.



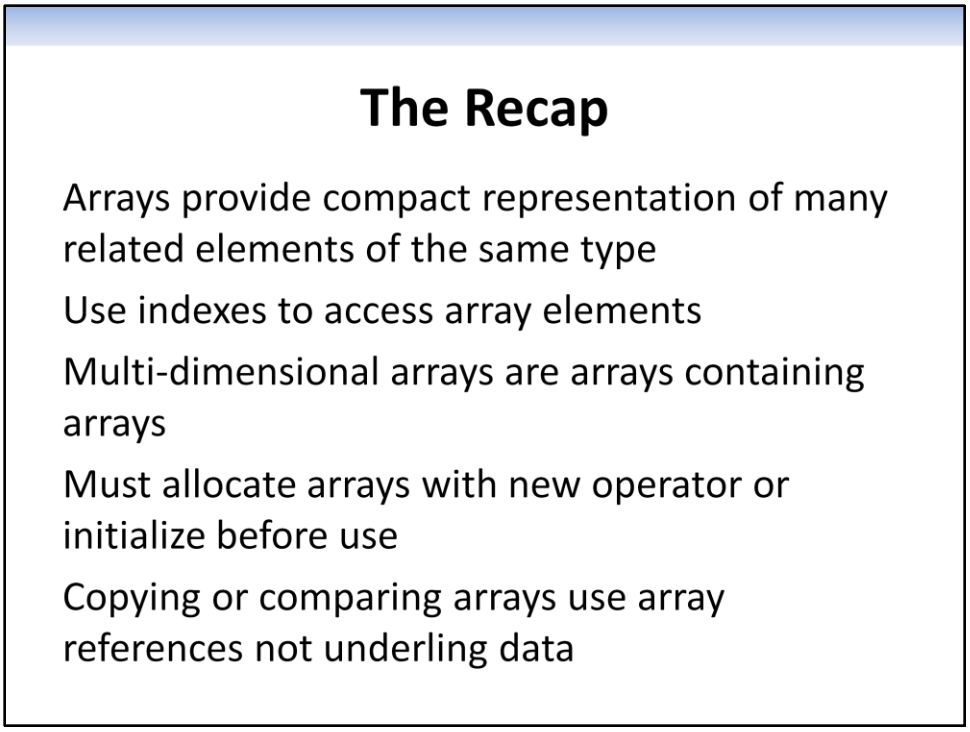
Arrays and Comparisons

- If comparing array elements of primitive types
  - Same as comparing to primitive type variables
    - Comparing the *values* of the elements
- If comparing array variables
  - Comparing the *reference* value, not data
  - If the *references* are different the result is false
    - Even if the underlining data is the same
    - This is true of all objects

Using arrays in comparisons takes some thought. If you are comparing the underlining elements of primitive type, it is the same as comparing two variables of that primitive type. The previous example demonstrated this.

If you are comparing array variables, you are comparing their *references*, not the underlining array data. If the references point to different arrays, the result will be false even if the underling data has the same values. This is true for all objects .

There has been a lot of information presented so lets review the highlights.



The Recap

- Arrays provide compact representation of many related elements of the same type
- Use indexes to access array elements
- Multi-dimensional arrays are arrays containing arrays
- Must allocate arrays with new operator or initialize before use
- Copying or comparing arrays use array references not underling data

OK here is the recap.

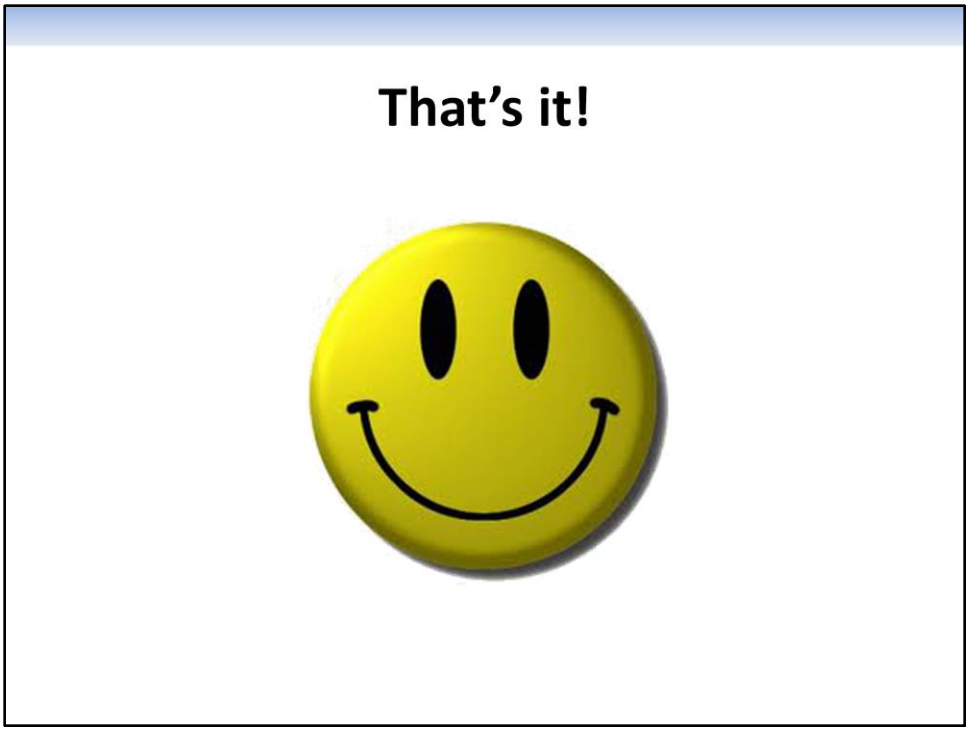
Arrays provide a compact representation of many related elements of the same type.

We use indexes to identify and access individual array elements.

Multi-dimensional arrays contain arrays within arrays. This starts to get fairly complicated beyond two dimensions.

Arrays must be allocated memory before they can be used. This can be done via the new operator or by initializing at time of array declaration.

When copying or comparing arrays, you are dealing with their references, not their underlining data.



That’s it!

That’s it!