This chapter describes how to create and use arrays in Java. Arrays are the simplest way to hold a series of values in Java.

The main limitation of arrays is that they are fixed size. Once you've created an array, you can't resize it later on. If you want resizability, you must use a collection class such as `ArrayList`; we'll cover collection classes later in the course.

Contents

1. Declaring and using arrays
2. Traversing arrays
3. Using the Arrays class
4. Multi-dimensional arrays

Demo project: **DemoArrays**

2

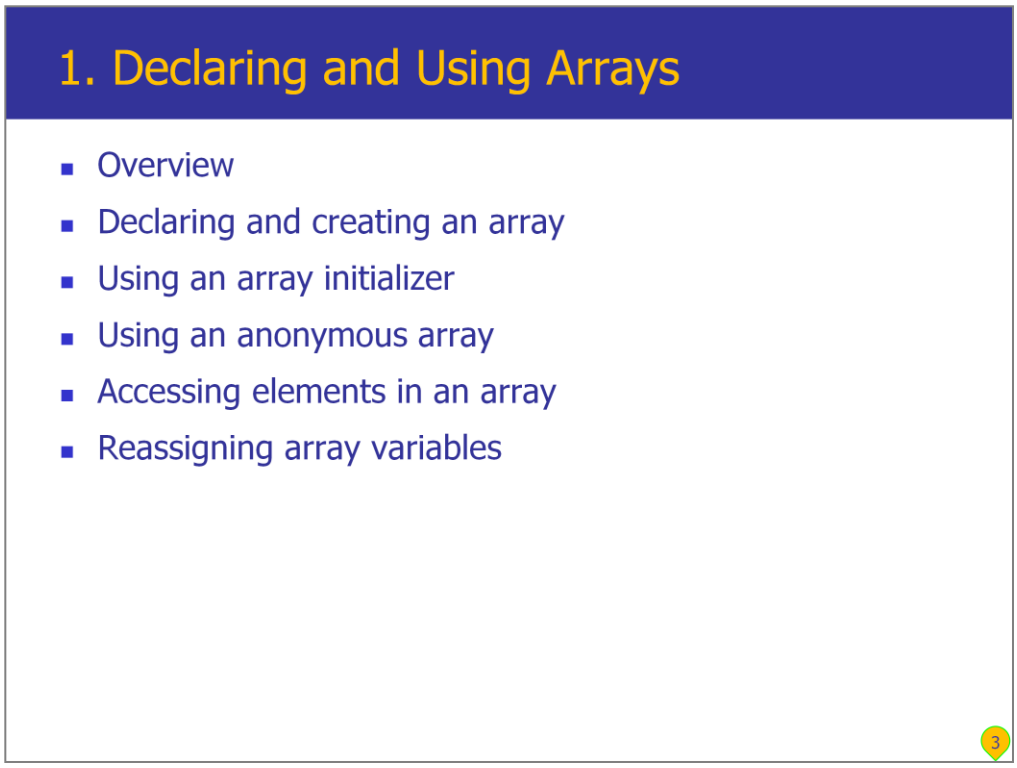
Section 1 introduces the basic syntax for arrays, where we show how to declare an array and access the elements inside the array.

Section 2 shows how to traverse the elements in an array, typically from start to finish. We'll also introduce a special variation of the `for` loop, which makes it easier to loop through elements in an array (or in a collection).

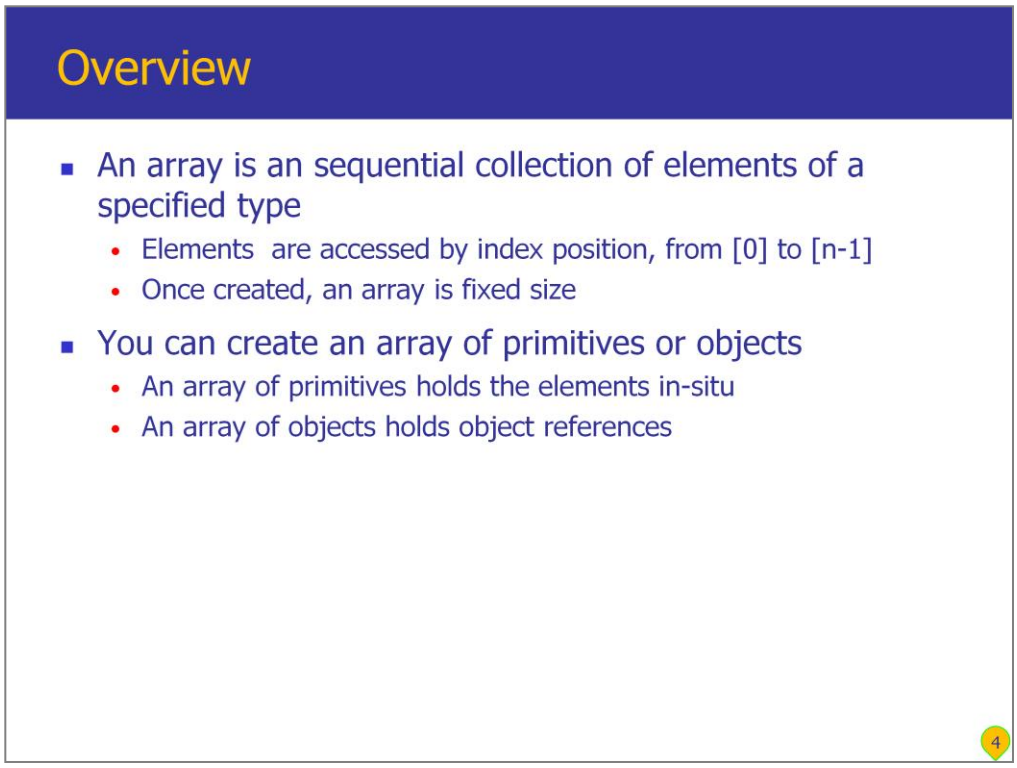
Section 3 introduces the `Arrays` class. This class contains some useful methods for manipulating arrays, e.g. sorting the elements.

Section 4 describes how to declare multi-dimensional arrays, e.g. a table with multiple rows and columns.

The demos for chapter are located in the `DemoArrays` project.



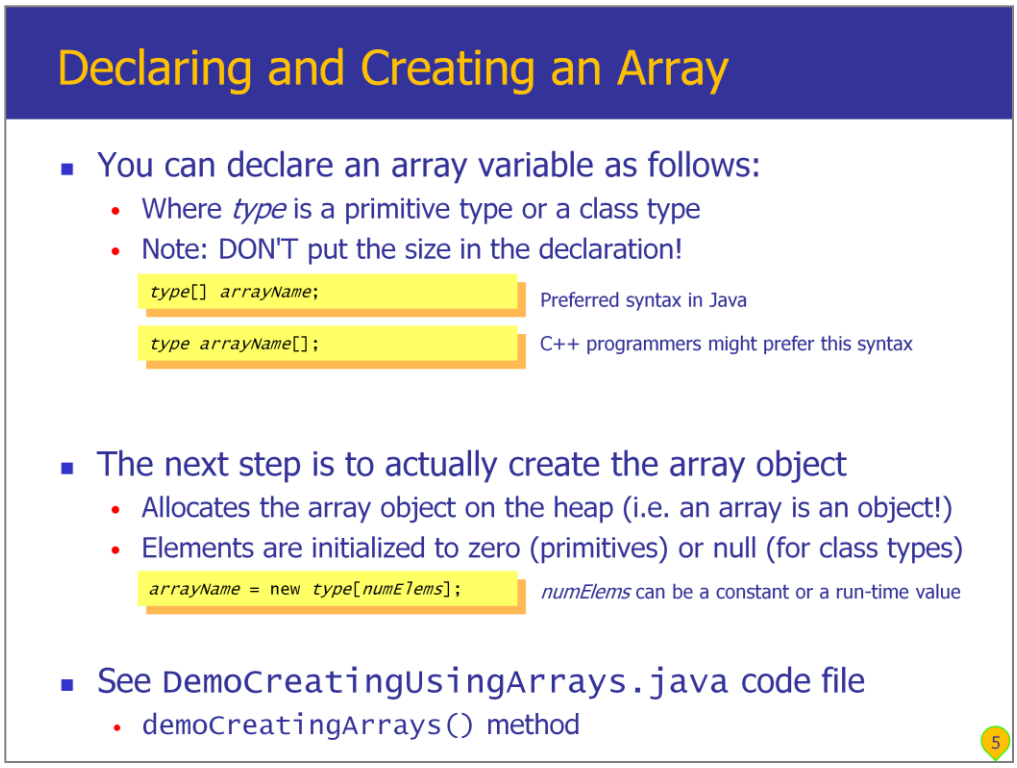
This section introduces the basic syntax for arrays, and focuses on some Java-specific rules that you might not expect of you're coming from another programming language.



An array is a fixed-size series of elements of the same type. You specify the size of the array when you create it, and the size is fixed thereafter. You access elements by index position, starting at 0 and running up to n-1.

You can create an array of primitives or an array of objects. Note the following points:

- If you create an array of primitives (e.g. an array of `int`s), the values are held in-situ within the array memory. All of the array elements are set to zero (or `false`, etc.) initially.
- If you create an array of objects (e.g. an array of `BankAccount`s), the array just contains object references - it's similar to the idea of an array of object pointers in C or C++. All of the array elements are set to `null` object references initially.



The upper code boxes in the slide show two different ways to declare an array variable. You can use either syntax, although the first syntax is generally preferred in Java. Note that you don't specify the array size inside the `[]`, because at this stage all you're doing is declaring a variable that is capable of pointing to an array; you haven't actually created the array itself yet.

Here are some examples of how to declare array variables:

```
int[] familyAges;
String[] planets;
```

Arrays in Java are objects. To create an array object, use the syntax shown in the bottom code box on the slide. Here are some examples of how to create array objects:

```
familyAges = new int[4]; // Contains 4 ints, all 0.
planets = new String[9]; // Contains 9 Strings, all null.
```

Note that the dimension between the `[]` can be a run-time expression. For example:

```
int familySize = … ;
familyAges = new int[familySize];
```

Using an Array Initializer

- You can declare, create, and populate an array all in one go, using array initializer syntax

```
type[] arrayName = { value0, value1, … };
```

- Example:

```
public static void demoArrayInitializers() {

  final int[] DAYS_IN_MONTH = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

  File[] logFiles = {
    new File("C:\\errors.log"),
    new File("C:\\access.log"),
    new File("C:\\audit.log")
  };

  …
}
```

DemoCreatingUsingArrays.java

6

If you want to create an array and initialize its elements immediately, you can specify a list of initialization values between `{}` when you create the array.

The first code box in the slide shows the general syntax, and the second code box shows some examples. Note the following points:

- `DAYS_IN_MONTH` contains 13 fixed integer values, indicating the number of days in each month. The first element is a placeholder, so the real elements are located at meaningful positions. For example, `DAYS_IN_MONTH[1]` gives the number of days in January, `DAYS_IN_MONTH[2]` indicates the number of days in February, etc.
- `logFiles` is an array of `File` objects. The array is initialized with three elements, where each element is a `File` object. For example, `logFiles[0]` is a `File` object representing the C:\errors.log file, and so on.

Using an Anonymous Array

- You can use array initializer syntax to initialize an array variable that already exists
  - ... by using an "anonymous array", as follows

```
type[] arrayName;

arrayName = new type[] { value0, value1, … };
```

- Example:

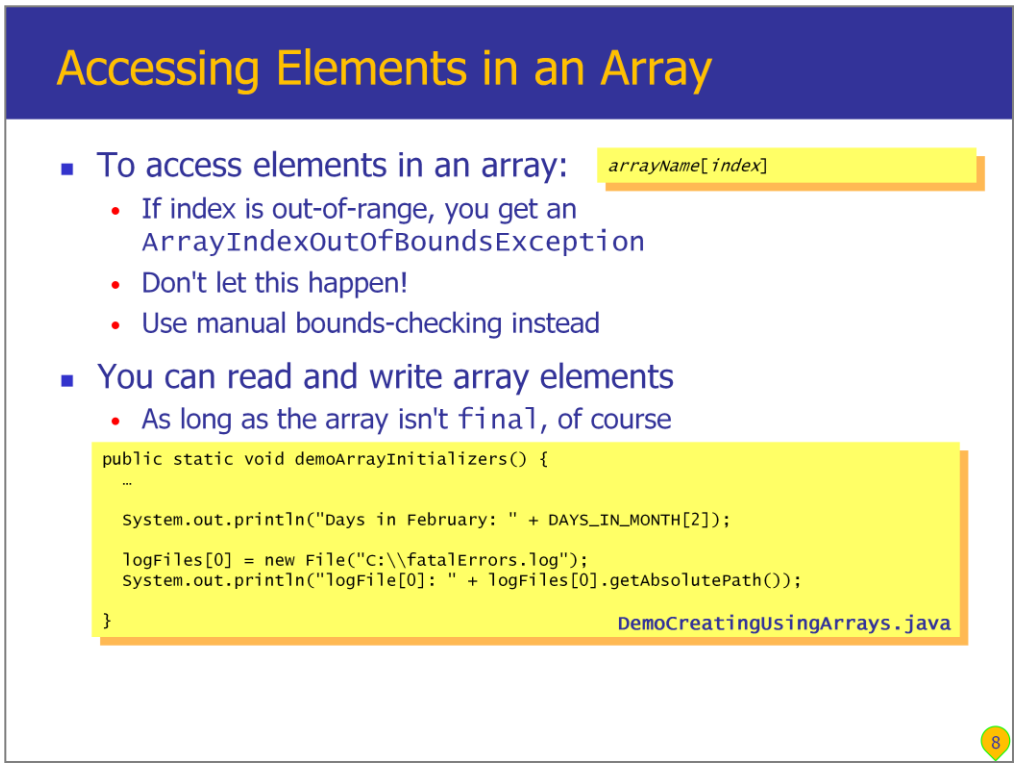
```
File[] logFiles;
…

logFiles = new File[] {
    new File("C:\\errors.log"),
    new File("C:\\access.log"),
    new File("C:\\audit.log")
};
```

7

The {} syntax shown on the previous slide is only valid at the point where you declare an array variable. If you've already declared an array variable beforehand, and then you want to assign an initialized array later on, you can use the anonymous array syntax shown in the slide.

Note the following points in the example in the second code box:

- First, we declare `logFiles` as an array variable, capable of pointing to an array of `File` objects.
- At some point later on, we create an actual `File` array and use {} to specify the initial values for the new array. The array will have three `File` objects, as you might expect. Note that the `new File[]` syntax is essential here - you couldn't just use the {} syntax to initialize the existing array variable directly.



To access elements in an array, use `[]` and specify an integer value between 0 and n-1 inclusive. For example, if you created an array with 10 elements, then the first element will be at position `[0]` and the last element will be at position `[9]`.

If you accidentally index outside the allowable range, you'll get a run-time exception. So be warned!

Reassigning Array Variables

- You can reassign an array variable to another array object
  - The other array must be a compatible type
  - The other array object must have the same number of dimensions
  - The other array object can be a different size
- Type-matching rules for primitive arrays:
  - You can't assign different primitive-type arrays
  - E.g. you can't assign a `byte[]` array object to an `int[]` variable
- Type-matching rules for reference-type arrays:
  - You can assign different reference-type arrays if "IS-A" rule applies
  - E.g. you can assign an `Employee[]` array object to a `Person[]` variable, if `Employee` inherits from `Person`

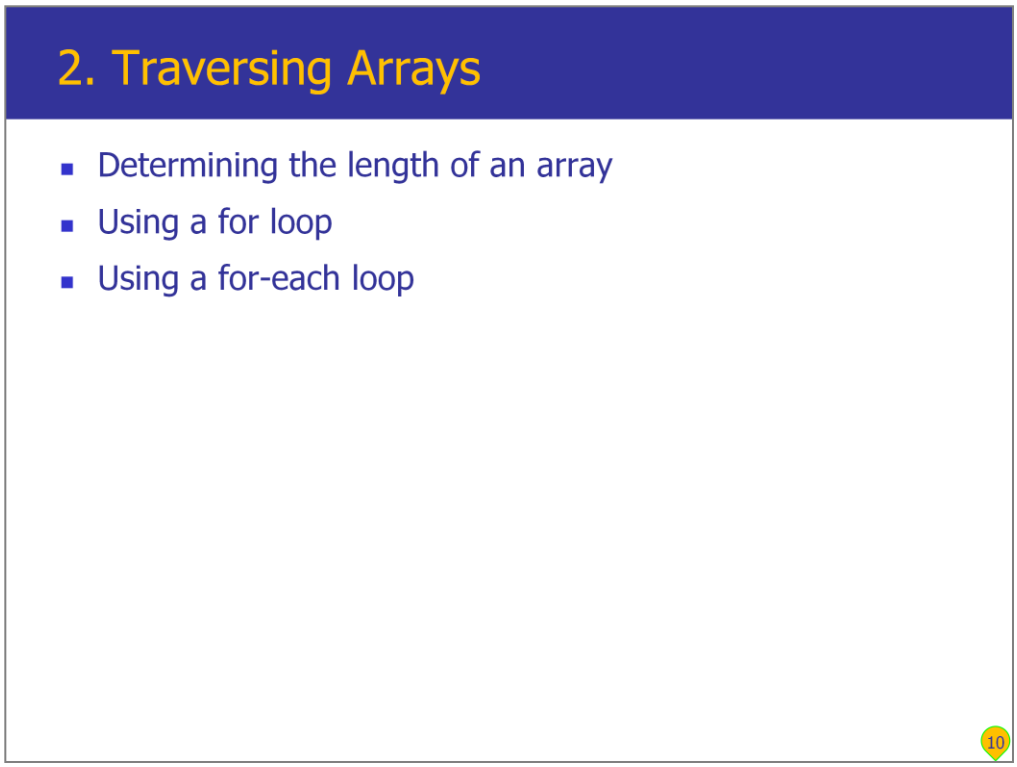
9

To conclude this section, we present some important rules about how to reassign array variables. For example, the following code is OK:

```
int[] array1 = { 10, 20, 30};
int[] array2 = { 100, 200, 300, 400, 500};
array1 = array2;
```

After this code has executed, `array1` will point to the same array object as does `array2`. In other words, `array[0]` is 100, `array[1]` is 200, etc.

The slide above also describes some additional rules and regulations. Some of these statements will make a bit more sense after we've covered multidimensional arrays and inheritance!



This section describes how to traverse the elements of an array, typically from start to end. We'll show how to do this using a regular for loop, and we'll also introduce Java's for-each loop as well.

Determining the Length of an Array

- The length of the array is determined via the `length` public field
  - Use this to manually check array indexes whenever the user has the chance to get it wrong!

```
public static void demoArrayLength() {

  final String[] DAYS_OF_WEEK = {"Sunday", "Monday", "Tuesday", "Wednesday",
                                 "Thursday", "Friday", "Saturday"};

  Scanner scanner = new Scanner(System.in);

  System.out.print("\nEnter a day number: ");
  int dayNumber = scanner.nextInt();

  if (dayNumber < 0 || dayNumber >= DAYS_OF_WEEK.length) {
    System.out.println("Day is out of range.");
  }
  else {
    System.out.println("Day " + dayNumber + " is " + DAYS_OF_WEEK[dayNumber]);
  }
}
```

DemoCreatingUsingArrays.java

11

Arrays have a length property that indicates the number of elements in an array. Thus the last element is at position `[anArray.length - 1]`.

You can use the length property to perform bounds checking, as shown in the example in the slide. This example ensures the user enters a day number between 0 and 6 inclusive.

Using a for Loop

- You can traverse an array using a for loop
  - This is the traditional approach
  - Use an integer array index that ranges from 0 to (n-1)

```
public static void demoArrayTraversal() {

  final String[] DAYS_OF_WEEK = {"Sunday", "Monday", "Tuesday", "Wednesday",
                                 "Thursday", "Friday", "Saturday"};

  for (int i = 0; i < DAYS_OF_WEEK.length; i++) {
    System.out.println("Day " + i + " is " + DAYS_OF_WEEK[i]);
  }
  …

}                                                    DemoCreatingUsingArrays.java
```

12

Given the fact arrays have a `length` property, you can write loops to iterate all the elements as shown in the slide. It's commonplace to use a `for` loop to do this.

Note that you don't have to iterate from start to end. You could start at the end and loop backwards if you prefer - it's your `for` loop, you can do whatever you like!

Using a for-each Loop

- From Java SE 1.5 onwards, you can use a for-each loop to iterate over an array (or collection)
  - Declaration part declares a variable that is compatible with elements in array/collection
  - Expression is an array/collection

```
for (declaration : expression) …
```

- Example:

```
public static void demoArrayTraversal() {

  final String[] DAYS_OF_WEEK = {"Sunday", "Monday", "Tuesday", "Wednesday",
                                 "Thursday", "Friday", "Saturday"};

  for (String dayOfWeek : DAYS_OF_WEEK) {
    System.out.println("Here's a day: " + dayOfWeek);
  }
  …

}                                                    DemoCreatingUsingArrays.java
```

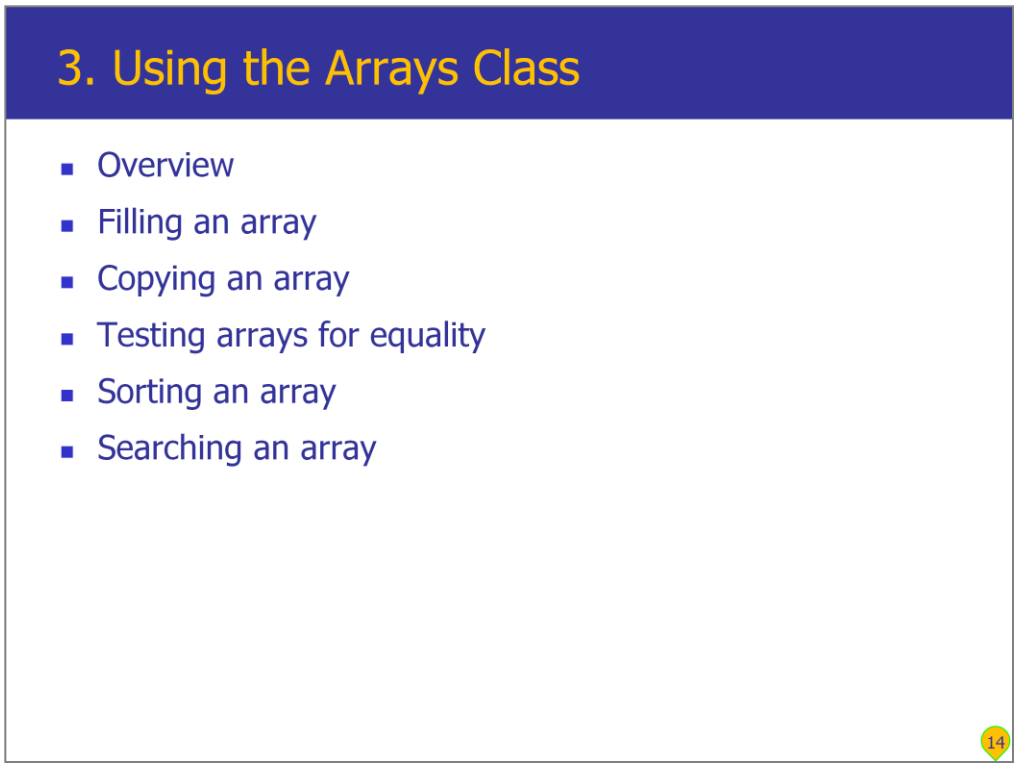
13

Java also provides a for-each loop, as shown here. You can use this construct to loop through all the elements in an array or a collection, starting at the beginning and visiting element through to the end.

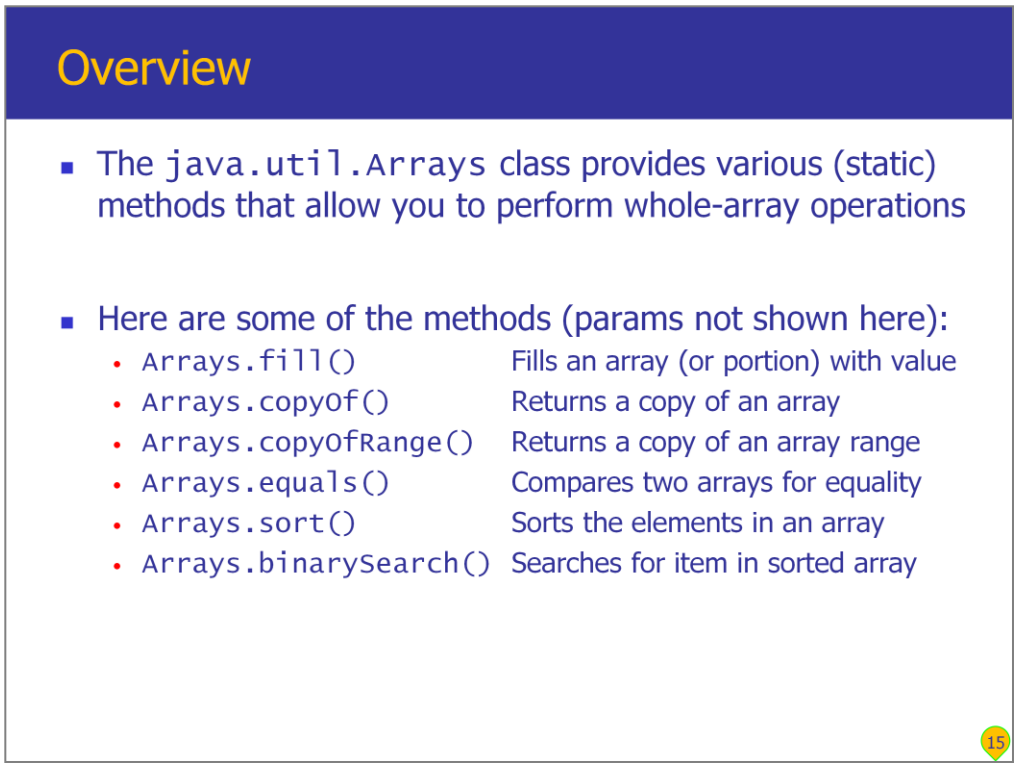
Note that there are some limitations when you use a for-each loop. Specifically, you can't use for-each if…

- You want to iterate backwards, skip some elements, etc.
- You want to replace elements while you are traversing an array/collection.
- You want to iterate over multiple arrays/collections in parallel.

In these circumstances, just use a regular `for` loop instead.



The `Arrays` class provides various useful methods for manipulating arrays, such as sorting, searching, and so on. These methods can save you from having to write a lot of manual code yourself.



The `Arrays` class is located in the `java.util` package, and provides a host of static methods for manipulating array objects. For full information about the `Arrays` class, see the following web site:

- http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html

Filling an Array

- `Arrays.fill(array, value)`
  - Fills an array with a specified value
- `Arrays.fill(array, index1, index2, value)`
  - Fills elements index1 to (index2-1) with a specified value

```
public static void demoFill() {

  int[] examMarks = { 89, 56, 82, 99, 72, 79, 64 };

  Arrays.fill(examMarks, 99);
  displayIntArray(examMarks, "Filled array with 99s.");

  Arrays.fill(examMarks, 2, 5, 100);
  displayIntArray(examMarks, "Filled array elements 2,3,4 with 100.");
}


private static void displayIntArray(int[] array, String msg) {
  System.out.println("\n" + msg);
  for(int elem: array) {
    System.out.println(elem);
  }
}
```

DemoArraysClass.java

16

This slide shows how to use the `Arrays` class to fill an array, via the `fill()` method. Like the majority of methods in the `Arrays` class, `fill()` is heavily overloaded to take different types of array and to initialize varying amounts of the array elements.

Copying an Array

- `Arrays.copyOf(array, length)`
  - Returns a copy of the array, truncated or padded if necessary
- `Arrays.copyofRange(array, index1, index2)`
  - Returns a copy of array elements index1 to (index2-1)

```
public static void demoCopy() {

  int[] examMarks = { 89, 56, 82, 99, 72, 79, 64 };

  int[] first3 = Arrays.copyOf(examMarks, 3);
  displayIntArray(first3, "Copy of first 3 elements.");

  int[] tenElems = Arrays.copyOf(examMarks, 10);
  displayIntArray(tenElems, "Copy of 10 elements.");

  int[] middle3Elems = Arrays.copyOfRange(examMarks, 2, 5);
  displayIntArray(middle3Elems, "Copy of middle 3 elements.");
}
```

DemoArraysClass.java

17

This slide shows how to use the `Arrays` class to extract a copy of some or all of the elements in an original array, via the `copyOf()` and `copyOfRange()` methods.

Testing Arrays for Equality

- `Arrays.equals(array1, array2)`
  - Returns `true` if arrays are of same type, and elements are equal to each other (as defined by calling `equals()` on element pairs)

```
public static void demoEquals() {

  int[] arr1 = { 89, 56, 82, 99, 72, 79, 64 };
  int[] arr2 = { 89, 56, 82, 99, 72, 79, 64 };
  int[] arr3 = { 11, 56, 82, 99, 72, 79, 64 };
  int[] arr4 = { 89, 56, 82, 99, 72, 79 };

  System.out.println("\narr1 equals arr2? " + Arrays.equals(arr1, arr2));
  System.out.println(  "arr1 equals arr3? " + Arrays.equals(arr1, arr3));
  System.out.println(  "arr1 equals arr4? " + Arrays.equals(arr1, arr4));

  String[] nephews1 = { "Huey", "Lewey", "Dewey" };
  String[] nephews2 = { "Huey", "Lewey", "Dewey" };

  System.out.println("nephews1 equals nephews2? " +
                       Arrays.equals(nephews1, nephews2));
}
```

DemoArraysClass.java

This slide shows how to use the `Arrays` class to determine if two arrays contain the same values, via the `equals()` method. Internally, this causes the `equals()` method to be called on corresponding elements in each array. This is handy if you have arrays of objects, because it allows you to define an `equals()` method in your class to define what equality actually means for your class.

Sorting an Array

- `Arrays.sort(array)`
  - Sorts the elements into ascending order
- `Arrays.sort(array, index1, index2)`
  - Sorts elements index1 to (index2-1)

```
public static void demoSort() {

  int[] examMarks = { 89, 56, 82, 99, 72, 79, 64 };

  Arrays.sort(examMarks, 2, 5);
  displayIntArray(examMarks, "Sorted array elements 2,3,4.");

  Arrays.sort(examMarks);
  displayIntArray(examMarks, "Sorted all elements.");
}
```

DemoArraysClass.java

19

This slide shows how to use the `Arrays` class to sort some or all of an array into ascending order, via the `sort()` method.

If you have an array of objects, you can supply a `Comparator` parameter to the `sort()` method, to define what it means for one object to be "greater than" another object. To find out more about `Comparator`, see the following web site:

- http://docs.oracle.com/javase/7/docs/api/java/util/Comparator.html

Searching an Array

- `Arrays.binarySearch(array, value)`
  - Searches a sorted array for a value, by using a binary search
  - Returns index of value if found, or -1 otherwise

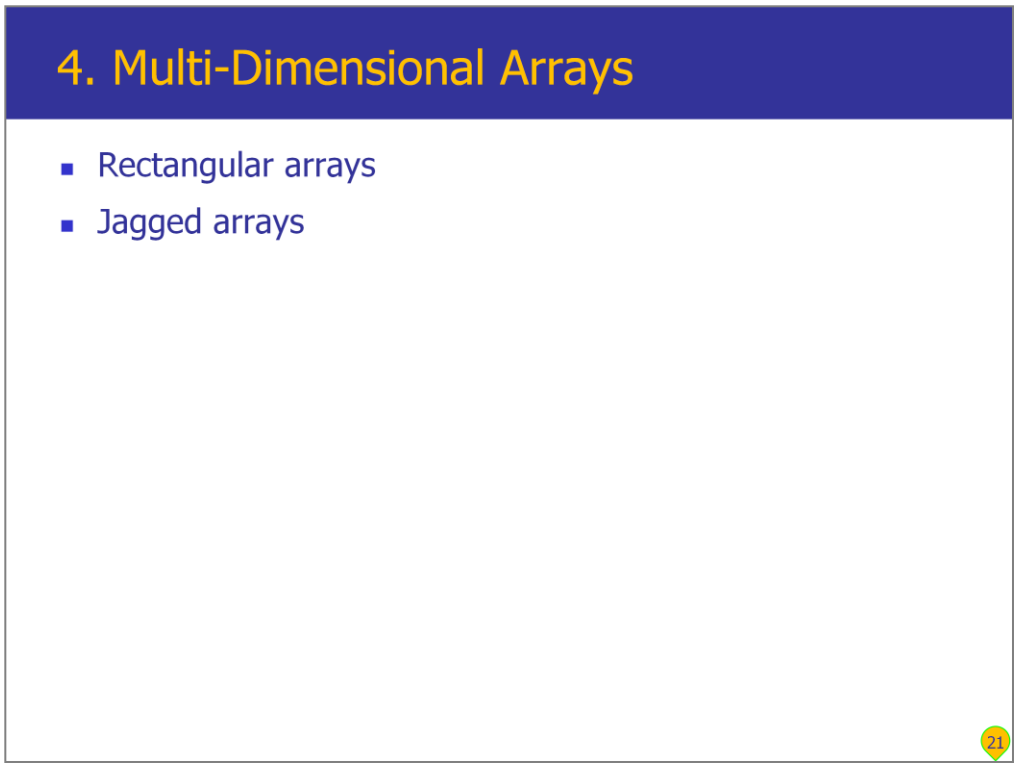
```
public static void demoSearch() {

  int[] examMarks = { 89, 56, 82, 99, 72, 79, 64 };

  Arrays.sort(examMarks);

  System.out.println("Index of 99: " + Arrays.binarySearch(examMarks, 99));
  System.out.println("Index of 22: " + Arrays.binarySearch(examMarks, 22));
}
```

DemoArraysClass.java

20

This slide shows how to use the `Arrays` class to search for a particular element in an array, via the `binarySearch()` method. The method returns the index of the matching element, or -1 if there is no match.

Note that `binarySearch()` only works if the array is already sorted. If you call `binarySearch()` on an unsorted array, the results are undefined.



All the arrays we've seen so far have been 1-dimensional, where we've declared and indexed using a single set of `[]`.

Java also allows you to define multidimensional arrays, e.g. for grids, cubes, etc. There are two kinds of multidimensional array:

- Rectangular arrays - Each row has the same number of columns
- Jagged arrays - Each row has its own number of columns (like an array of arrays)

Rectangular Arrays

- To create a rectangular array (e.g. with 2 dimensions):

```
type[][] arrayName = new type[numRows][numCols];
```

- You can use initializer syntax, with multiple sets of braces:

```
type[][] arrayName =
{
  { …, …, … },
  { …, …, … },
    …
};
```

- Use nested loops to process:

```
for (int r = 0; r < arrayName.length; r++) {
  for (int c = 0; c < arrayName[r].length; c++) {
    // Access element arrayName[r][c]
  }
}
```

DemoMultiDimArrays.java

22

In a rectangular array, each row has the same number of columns.

To declare a rectangular array, use a separate set of `[]` for each dimension and specify the number of elements in that dimension. The following example declares a 2-D array of integers, where there are 5 rows and each row has 10 columns:

```
int[][] myRectArray = new int[5][10];
```

To access an element in this 2-D array, use two separate `[]` to specify the row and column you want. For example, to access the element in row 2 and column 6:

```
int value = myRectArray[2][6];
```

If you want to traverse all the elements in the array, you can use nested loops as shown in the lower code box in the slide. Notice the use of the `length` property to determine the number of rows in the array, and the number of columns in each row.

Jagged Arrays

- To create a jagged array:
  - Specify number of rows initially, but not number of columns
  - Specify number of columns later, on a row-by-row basis

```
type[][] arrayName = new type[numRows][];
…
arrayName[0] = new type[numColsForRow0];
arrayName[1] = new type[numColsForRow1];
arrayName[2] = new type[numColsForRow2];
…
```

DemoMultiDimArrays.java

- You can use initializer syntax, with multiple sets of braces
  - Same previous slide
- Use nested loops to process
  - See previous slide

23

In a jagged array, each row has its own number of columns. This is useful if each row needs to store a different amount of data - it avoids unnecessary empty elements at the end of sparse rows, which you'd get if you created a rectangular array.

To declare a jagged array, use a separate set of `[]` for each dimension as before, but this time omit the number of elements in the final dimension. This effectively creates an array of array pointers. For each row in the array, you must explicitly allocate another array to hold the columnar elements for that row. In this way, each row can decide to allocate a different number of elements. The code box in the slide shows an example of how to do this.

The syntax for initializing and traversing jagged arrays is the same as for rectangular arrays on the previous slide.

Any Questions?