# Lecture 13 & 14

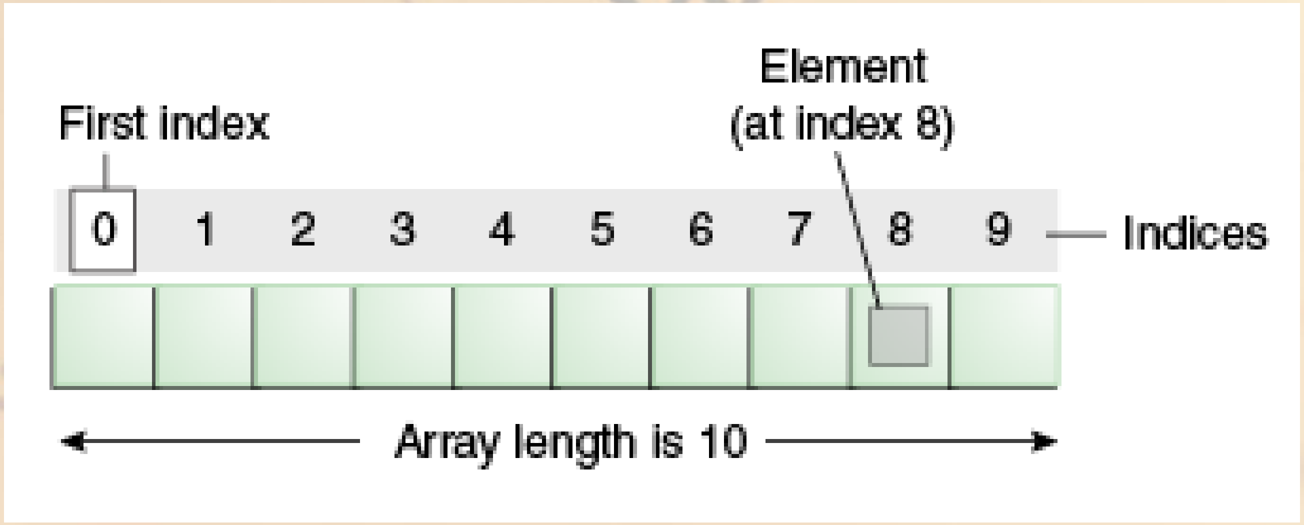
## Single Dimensional Arrays

Dr. Martin O'Connor
CA166
www.computing.dcu.ie/~moconnor

## Table of Contents

# Arrays

- An array is a container object that holds a fixed number of values of a single type
- The length of the array is established when the array is created. After creation, its length is fixed
- An Example:



# Arrays

- Arrays are used to store and manipulate a collection of related values
- It would be impractical to use a sequence of variables such as value1, value2, value3 and so on
- An array is a sequence of variables of the same data type.
- The data type can be any of Java's primitive types (*int*, *short*, *byte*, *long*, *float*, *double*, *boolean*, *char*), or a class.
- Each item in an array is called an element.
- Each element is accessed by its numerical index

## Declaring and Instantiating Arrays

Arrays are objects, so creating an array requires two steps:

1. declaring a reference to the array
2. instantiating the array

To declare a reference to the array, use this syntax:

```
datatype[] arrayName;
```

To instantiate an array, use this syntax:

```
arrayName = new datatype[size];
```

where *size* is an expression that evaluates to an integer and specifies the number of elements in the array.

## Examples

Declaring arrays:

```
double[] dailyTemps; // elements are doubles
String[] cdTracks;   // elements are Strings
boolean[] answers;   // elements are booleans
Auto[] cars;  // elements are Auto references
int[] cs101, bio201; // two int arrays
```

Instantiating these arrays:

```
dailyTemps = new double[365];  // 365 elements
cdTracks = new String[15];     // 15 elements
int numberOfQuestions = 30;
answers = new boolean[numberOfQuestions];
cars = new Auto[3];            // 3 elements
cs101 =  new int[5];           // 5 elements
bio201 = new int[4];           // 4 elements
```

## The *Auto* Class

- We will be using the Auto class as a simple example throughout this lecture.
- The *Auto* class has three instance variables: *model*, *milesDriven*, and *gallonsOfGas*

```
public class Auto
{
   private String model;
   private int milesDriven;
   private double gallonsOfGas;
}
```

# Default Values for Elements

When an array is instantiated, the elements are assigned default values according to the array data type.

| **Array data type** | **Default value** |
|---|---|
| *byte, short, int, long* | 0 |
| *float, double* | 0.0 |
| *char* | The null character |
| *boolean* | *false* |
| Any object reference (for example, a *String*) | *null* |

## Combining the Declaration and Instantiation of Arrays

- One way to create an array is with the *new* operator

Syntax:

```
 datatype[] arrayName  = new datatype[size];
```

Examples:

```
double[] dailyTemps = new double[365];
String[] cdTracks = new String[15];
int numberOfQuestions = 30;
boolean[] answers = new boolean[numberOfQuestions];
Auto[] cars = new Auto[3];
int[] cs101 = new int[5], bio201 = new int[4];
```

## Assigning Initial Values to Arrays

Arrays can be instantiated by specifying a list of initial values.

Syntax:

```
datatype[] arrayName =  { value0, value1, … };
  where valueN is any expression evaluating to
   the data type of the array and is the value
   to assign to the element at index N.
```

Examples:

```
int nine = 9;
int[] oddNumbers = { 1, 3, 5, 7, nine, nine + 2,
                     13, 15, 17, 19 };
Auto sportsCar = new Auto( "Ferrari", 0, 0.0 );
Auto[] cars = { sportsCar, new Auto(),
                new Auto("BMW", 100, 15.0 ) };
```

## Common Error Trap

- An initialization list can be given only when the array is declared.
  - Attempting to assign values to an array using an initialization list after the array is instantiated will generate a compiler error.
- The *new* keyword is not used when an array is instantiated using an initialization list. Also, no size is specified for the array; the number of values in the initialization list determines the size of the array.

## Common Error Trap

- When declaring an array, you can also place the brackets after the array's name

```
double dailyTemps[];    // elements are doubles
```

- **!DO NOT DO THIS!**
- The Java programming convention strongly discourages this style of declaration.
- Why?  brackets are used to both indicate and identify an array type and consequently, should always appear beside the type declaration.

```
double[] dailyTemps;    // elements are doubles
```

## Accessing Array Elements

- To access an element of an array, use this syntax:

```
arrayName[exp]
```

```
 where exp is an expression that
 evaluates to an integer.
```

- *exp* is the element's index — its position within the array.
- The index of the first element in an array is 0.
- *length* is a public, constant integer instance variable that holds the number of elements in the array and is accessed using this syntax:

```
arrayName.length
```

- Example: `dailyTemps.length`
-

## Common Error Trap

Attempting to access an element of an array using an index less than 0 or greater than *arrayName.length - 1* will generate an *ArrayIndexOutOfBoundsException* at run time.

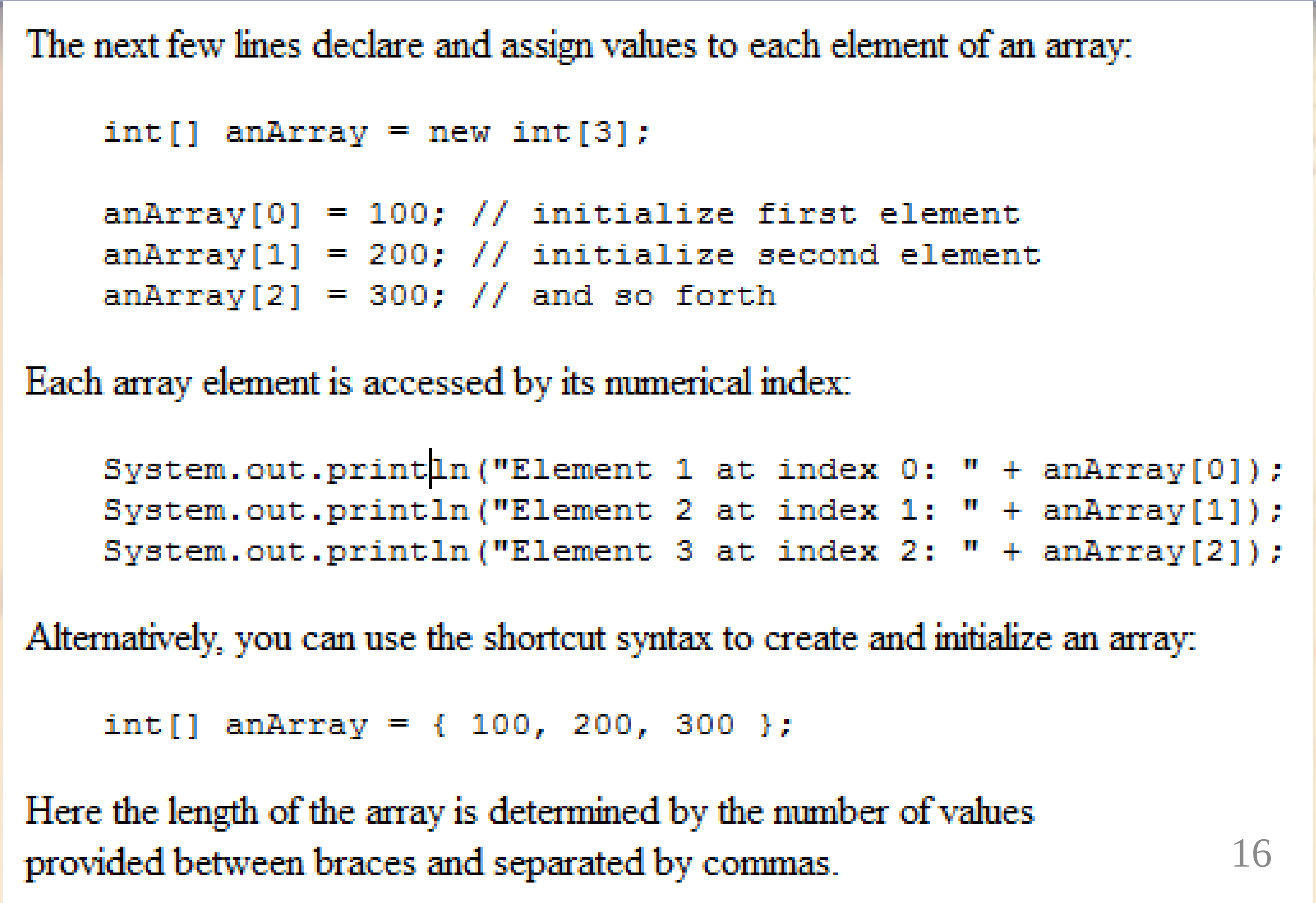
Note that for an array, *length* – without parentheses – is an instance variable, whereas for *Strings*, *length( )* – with parentheses – is a method.

Note also that the array's instance variable is named *length*, rather than *size*.

## Accessing Array Elements

| Element | Syntax |
|---|---|
| Element 0 | `arrayName[0]` |
| Element *i* | `arrayName[i]` |
| Last element | `arrayName[arrayName.length – 1]` |

# Accessing Array Elements



The next few lines declare and assign values to each element of an array:

```
int[] anArray = new int[3];

anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
anArray[2] = 300; // and so forth
```

Each array element is accessed by its numerical index:

```
System.out.println("Element 1 at index 0: " + anArray[0]);
System.out.println("Element 2 at index 1: " + anArray[1]);
System.out.println("Element 3 at index 2: " + anArray[2]);
```

Alternatively, you can use the shortcut syntax to create and initialize an array:

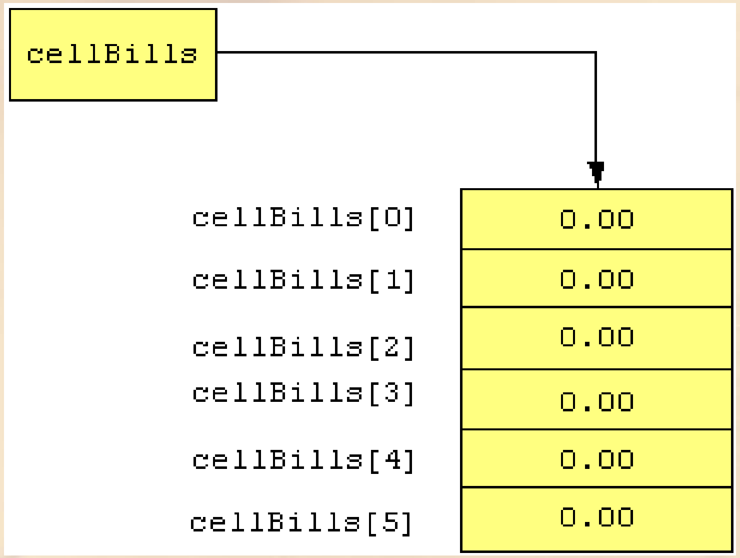
```
int[] anArray = { 100, 200, 300 };
```

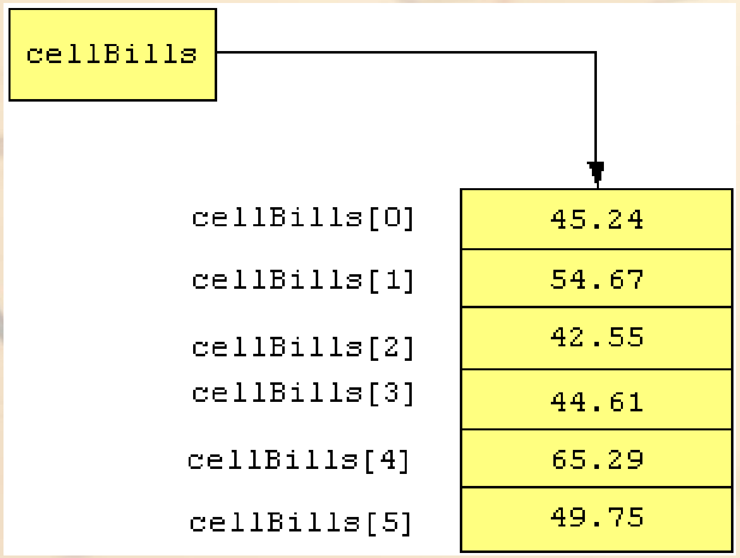
Here the length of the array is determined by the number of values provided between braces and separated by commas.

# Another Example *cellBills* Array

When instantiated:

After assigning values:





## Instantiating an Array of Objects

To instantiate an array with a class data type:

1. instantiate the array (elements are object references, initialized to *null*)
2. instantiate the objects

Example:

```
// instantiate array; all elements are null
Auto[] cars = new Auto[3];
// instantiate objects and assign to elements
Auto sportsCar = new Auto( "Miata", 100, 5.0 );
cars[0] = sportsCar;
cars[1] = new Auto();
// cars[2] is still null
```

## Aggregate Array Operations

We can perform the same operations on arrays as we do on a series of input values.

- calculate the total of all values
- count values meeting specified criteria
- find the average value
- find a minimum or maximum value, etc.

To perform an operation on all elements in an array, we use a *for* loop to perform the operation on each element in turn.

## Standard *for* Loop Header for Array Operations

```
for (int i = 0; i < arrayName.length; i++)
```

- initialization statement ( `int i = 0` ) creates index *i* and sets it to the first element ( 0 ).
- loop condition ( `i < arrayName.length` ) continues execution until the end of the array is reached.
- loop update ( `i++`) increments the index to the next element, so that we process each element in order.

Inside the *for* loop, we reference the current element as:

```
arrayName[i]
```

## Printing All Elements of an Array

Example: This code prints each element in an array named *cellBills*, one element per line (assuming that *cellBills* has been instantiated as an array of doubles):

```
for (int i = 0; i < cellBills.length; i++)
{
   System.out.println( cellBills[i] );
}
```

## Reading Data Into an Array

Example: this code reads values from the user into an array named *cellBills*, which has previously been instantiated:

```
Scanner scan = new Scanner( System.in );
for (int i = 0; i < cellBills.length; i++)
{
   System.out.print( "Enter bill > "  );
   cellBills[i] = scan.nextDouble( );
}
```

- *Note: the above code has no error checking*

## Calculating a Total

Example: this code calculates the total value of all elements in an array named *cellBills*, which has previously been instantiated:

```
double total = 0.0; // initialize total
for (int i = 0; i < cellBills.length; i++ )
{
  total += cellBills[i];
}
System.out.println( "The total is " + total );
```

## Finding Maximum/Minimum Values

Example: this code finds the index of the maximum value in an array named *cellBills*:

```
// make first element the current maximum
int maxIndex= 0;

// start for loop at element 1
for (int i = 1; i < cellBills.length; i++ )
{
  if ( cellBills[i] > cellBills[maxIndex] )
      maxIndex = i;
}
System.out.println( "The maximum is "
                    + cellBills[maxIndex] );
```

## Copying Arrays

Suppose we want to copy the elements of an array to another array. We could try this code:

```
double[] billsBackup = new double [6];
billsBackup = cellBills; // incorrect!
```

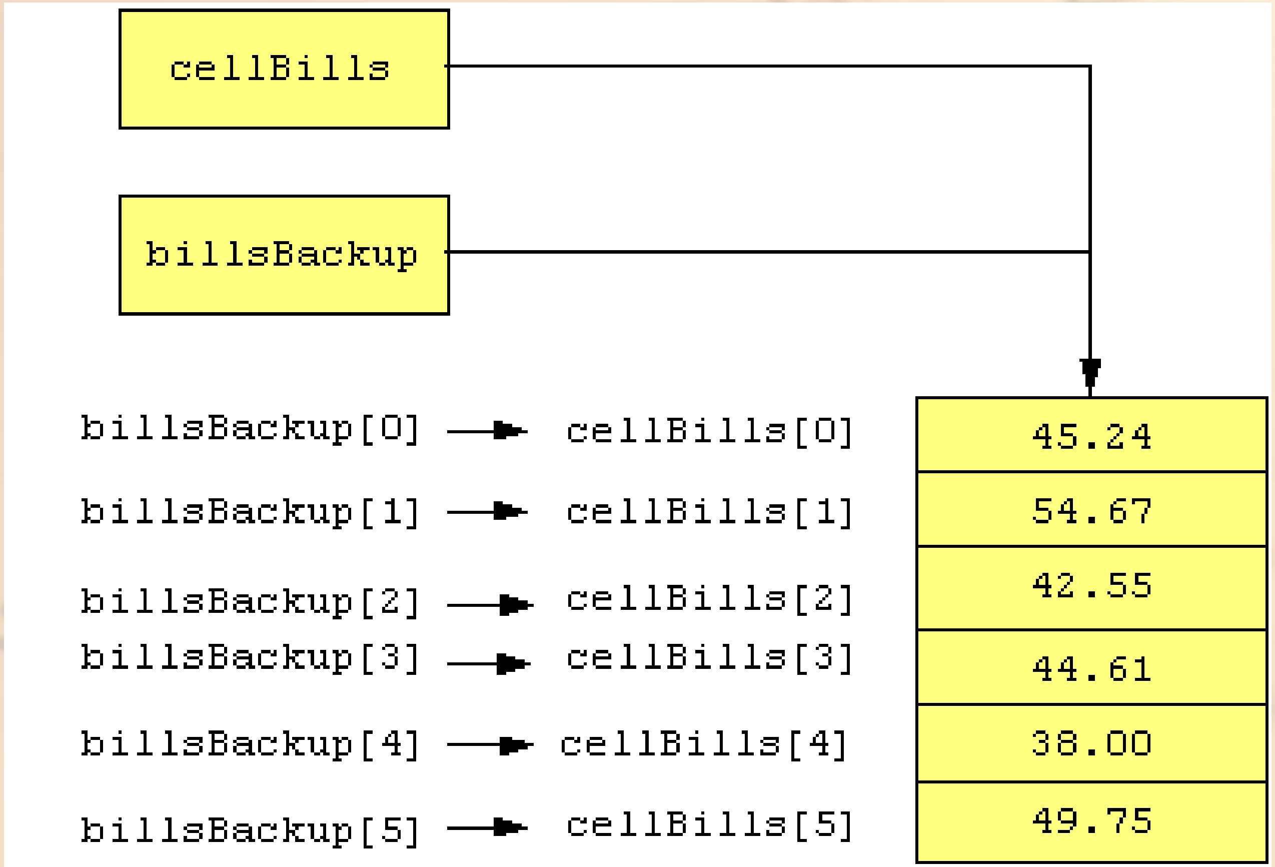
Although this code compiles, it is logically incorrect! We are copying the *cellBills* object reference to the *billsBackup* object reference. We are not copying the array data.

The result of this code is shown on the next slide ->

# Copying Array References

```
billsBackup = cellBills;
```

The line of code above line has this effect. Both references point to the same array.


cellBills
billsBackup
billsBackup[0] → cellBills[0]
billsBackup[1] → cellBills[1]
billsBackup[2] → cellBills[2]
billsBackup[3] → cellBills[3]
billsBackup[4] → cellBills[4]
billsBackup[5] → cellBills[5]
45.24
54.67
42.55
44.61
38.00
49.75

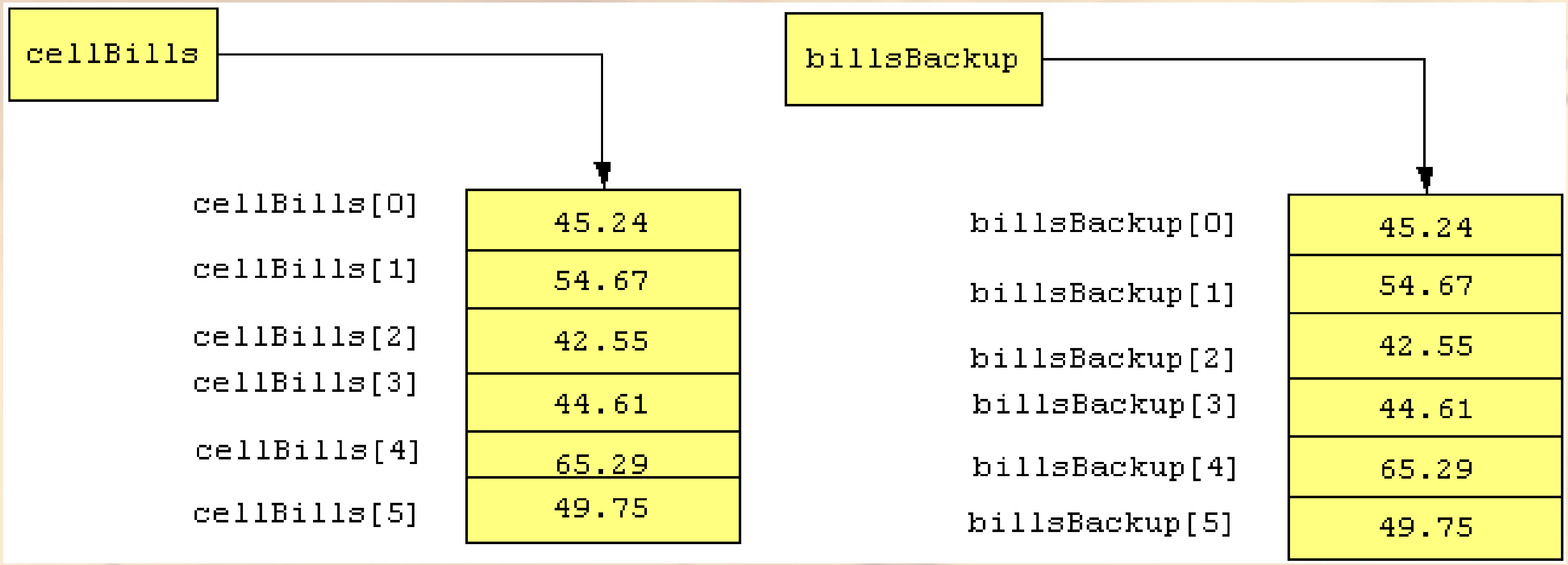
## Copying Array Values

Example: this code copies the values of all elements in an array named *cellBills* to an array named *billsBackup*, both of which have previously been instantiated with the same *length*:

```
for (int i = 0; i < cellBills.length; i++)
{
   billsBackup[i] = cellBills[i];
}
```

The effect of this *for* loop is shown on the next slide ->

# Copying Array Values



*A separate copy of the array has been created.*

## Changing an Array's Size

An array's *length* instance variable is constant.

- that is, arrays are assigned a constant size when they are instantiated.

To expand an array while maintaining its original values:

1. Instantiate a new array with the new size and a temporary name.
2. Copy the original elements to the new array.
3. Then point the original array reference to the new array.
4. Assign a *null* value to the temporary array reference.

## Expanding the Size of an Array

This code expands the size of the *cellBills* array from 6 elements to 12 elements:

```
//instantiate new array with a temporary name
double[] temp = new double[12];

// copy all elements from cellBills to temp
for (int i = 0; i < cellBills.length; i++)
{
  temp[i] = cellBills[i]; // copy each element
}

// point cellBills to new array
cellBills = temp;
temp = null;  // assign null to temp reference
```

## Comparing Arrays for Equality

To compare whether the elements of two arrays are equal:

1. Determine if both arrays have the same length.
2. Compare each element in the first array with the corresponding element in the second array.

To do this, we'll use a flag variable and a *for* loop.

At the end of the code, the value of the flag variable *isEqual* will indicate whether the arrays are equal.

## Comparing *cellBills1* to *cellBills2*

```
boolean isEqual = true;     // flag variable

if (cellBills1.length != cellBills2.length ) {
    isEqual = false;        // sizes are different
} else {
  for (int i = 0; i < cellBills1.length && isEqual; i++ )
  {
     if (Math.abs(cellBills1[i] - cellBills2[i]) > 0.001)
     {
         isEqual = false; // elements are not equal
     }
  }
}
```

## Using Arrays in Classes

In a user-defined class, an array can be

- an instance variable
- a parameter to a method
- a return value from a method
- a local variable in a method

# Methods with Array Parameters

To define a method that takes an array as a parameter, use this syntax:

```
accessModifier returnType methodName(
                              dataType[] arrayName )
```

To define a method that returns an array, use this syntax:

```
accessModifier dataType[] methodName(
                              parameterList )
```

To pass an array as an argument when calling a method, use the array name without brackets:

```
methodName( arrayName )
```

## Common Error Trap

If you think of the brackets as being part of the data type of the array, then it's easy to remember that

- brackets are included in the method header (or method signature) where the data types of parameters are given
- brackets are not included in method calls (where the data itself is given).

## Array Instance Variables

A constructor (or mutator method) that accepts an array parameter should instantiate an instance variable array and copy the elements from the parameter array to the instance variable.

```
// constructor
public CellPhone(double[] bills )
{
  // instantiate instance variable array
  // with same length as parameter
  cellBills = new double[bills.length];

  // copy parameter array bills to cellBills array
  for (int i = 0; i < cellBills.length; i++) {
     cellBills[i] = bills[i];
   }
}
```

# Accessors for Arrays

Similarly, an accessor method for the array instance variable should return a reference to a copy of the array.

```
public double[] getCellBills( )
{
  // instantiate temporary array
  double[] temp = new double[cellBills.length];

  // copy instance variable values to temp
  for (int i = 0; i < cellBills.length; i++)
       temp[i] = cellBills[i];

  // return copy of array
  return temp;
}
```

## SOFTWARE ENGINEERING TIP

- Sharing array references with the client violates encapsulation.
  - To accept an array as a parameter to a method, instantiate an instance variable array and copy the elements of the parameter array to the instance variable.
  - Similarly, to return an instance variable array, a method should copy the elements of the instance variable array to a temporary array and return a reference to the temporary array.

# Copying Arrays

- The System class has an *arraycopy()* method that you can use to efficiently copy data from one array to another

```
public static void arraycopy(Object src, int srcPos,
                             Object dest, int destPos, int length)
```

- The two Object arguments specify the array to copy *from* and the array to copy *to.*

- The three int arguments specify the starting position in the source array, the starting position in the destination array, and the number of array elements to copy.

## Copying Arrays

- The following program, *ArrayCopyDemo*, declares an array of char elements, spelling the word "decaffeinated." It uses the System.arraycopy() method to copy a subsequence of array components into a second array

```
class ArrayCopyDemo {
    public static void main(String[] args) {
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',
                            'i', 'n', 'a', 't', 'e', 'd' };
        char[] copyTo = new char[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);
        System.out.println(new String(copyTo));
    }
}
```

The output from this program is:

```
    caffein
```

## Array Manipulations

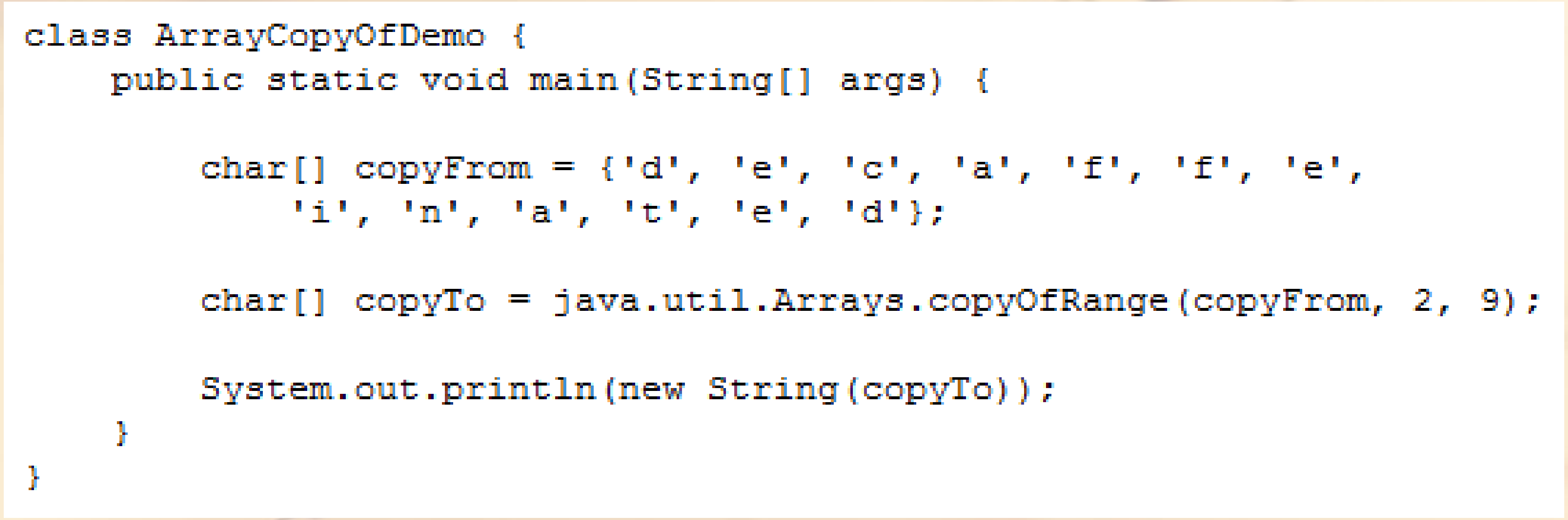
- Arrays are a powerful and useful concept used in programming
- Java SE provides methods to perform some of the most common manipulations related to arrays
- As we have seen, the *ArrayCopyDemo* example uses the *arraycopy()* method of the System class instead of manually iterating through the elements of the source array and placing each one into the destination array
- This is performed behind the scenes, enabling the developer to use just one line of code to call the method

## Array Manipulations

- For your convenience, Java SE provides several methods for performing array manipulations (common tasks, such as copying, sorting and searching arrays) in the java.util.Arrays class.
- The previous example can be modified to use the CopyOfRange() method of the java.util.Arrays class
- The difference is that using the CopyOfRange() method does not require you to create the destination array before calling the method, because the destination array is returned by the method

## Array Manipulations

- The output from this program is the same (caffein), although it requires fewer lines of code.



```
class ArrayCopyOfDemo {
    public static void main(String[] args) {

        char[] copyFrom = {'d', 'e', 'c', 'a', 'f', 'f', 'e',
            'i', 'n', 'a', 't', 'e', 'd'};

        char[] copyTo = java.util.Arrays.copyOfRange(copyFrom, 2, 9);

        System.out.println(new String(copyTo));
    }
}
```

## Array Manipulations

Some other useful operations provided by methods in the java.util.Arrays class, are:

- Searching an array for a specific value to get the index at which it is placed (the binarySearch() method).
- Comparing two arrays to determine if they are equal or not (the equals() method).
- Filling an array to place a specific value at each index (the fill() method).
- Sorting an array into ascending order using the sort() method

## Retrieving Command Line Arguments

The syntax of an array parameter for a method might look familiar. We've seen it repeatedly in the header for the *main* method:

```
public static void main(String[] args )
```

*main* receives a *String* array as a parameter. That array holds the arguments, if any, that the user sends to the program from the command line.

For example, command line arguments might be:

- the name of a file for the program to use
- configuration preferences

## Printing Command Line Arguments

```
public static void main(String[] args)
{
   System.out.println( "The number of parameters "
         + " is " + args.length );

   for (int i = 0; i < args.length; i++)
   {
      System.out.println( "args[" + i + "]: "
            + args[i] );
   }
}
```