Chapter 8

Single-Dimensional Arrays

Topics

- Declaring and Instantiating Arrays
- Accessing Array Elements
- Writing Methods
- Aggregate Array Operations
- Using Arrays in Classes
- Searching and Sorting Arrays
- Using Arrays as Counters

Arrays

- 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 variable in the array is an **element.**
- We use an **index** to specify the position of each element in the array.
- Arrays are useful for many applications, including calculating statistics or representing the state of a game.

Declaring and Instantiating Arrays

- Arrays are objects, so creating an array requires two steps:
  1. declaring the 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.

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 integer 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
cs201 =  new int[5];           // 5 elements
bio101 = new int[4];           // 4 elements
```


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	space
boolean	*false*
Any object reference (for example, a *String*)	*null*

Combining the Declaration and Instantiation of Arrays

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 an 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.

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 read-only integer instance variable that holds the number of elements in the array and is accessed using this syntax:

```
arrayName.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*.

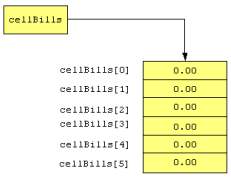
Accessing Array Elements

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

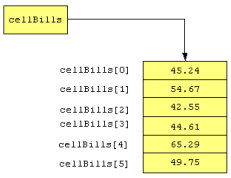
- *See Example 8.1 CellBills.java*

cellBills Array

When instantiated: values:



After assigning values:



Instantiating an Array of Objects

- To instantiate an array with a class data type:
  1. instantiate the array
  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
```

- *See Example 8.2 AutoArray.java*

Aggregate Array Operations

- We can perform the same operations on arrays as we do on a series of input values.
  - calculate the total value
  - 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):

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

- *See **Example 8.3** PrintingArrayElements.java*

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( );
}
```

- *See **Example 8.4** ReadingDataIntoAnArray.java*

Summing the Elements of an Array

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

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

- *See Example 8.5 SummingArrayElements.java*

Finding Maximum/Minimum Values

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

```
// make first element the current maximum
double maxValue = cellBills[0];
// start for loop at element 1
for (int i=1;i<cellBills.length; i++)
{
  if ( cellBills[i] > maxValue )
      maxValue = cellBills[i];
}
System.out.println( "The maximum is "
+
                        maxValue );
```

- *See Example 8.6 MaxArrayValue.java*

Copying Arrays

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

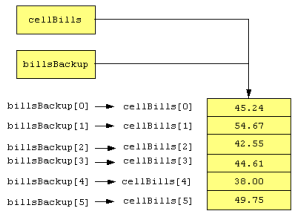
```
double [] billsBackup = new double [6];
billsBackup = cellBills; // incorrect!
```

- 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;
```

has this effect:



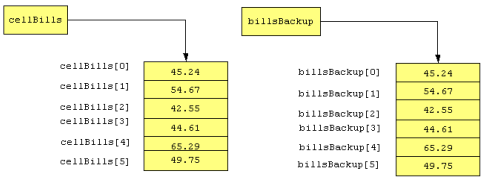
Copying Array Values

- 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.
- *See **Example 8.7** CopyingArrayElements.java*

Copying Array Values



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 an array with the new size and a temporary name.
  2. Copy the original elements to the new array.
  3. 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 will expand the size of the *cellBills* array from 6 to 12 elements:

```
//instantiate new array
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;
```


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.

Comparing *cellBills1* to *cellBills2*

```
boolean isEqual = true;
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
  }
}
```

- *See **Example 8.8** ComparingArrays.java*

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

Arrays as Instance Variables

- Because arrays are objects, the name of an array is an object reference.
- Methods must be careful not to share references to instance variables with the client. Otherwise, the client could directly change the array elements using the reference to the array instance variable.
- *See Examples 8.11 & 8.12*

Array Instance Variables

- A constructor (or mutator) that accepts an array as a parameter should instantiate a new array for the instance variable and copy the elements from the parameter array to the instance variable array.

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

  // copy parameter array 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 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 values for an instance variable array as a parameter to a method, instantiate a new array and copy the elements of the parameter array to the new array.
- 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.

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] );
  }
}
```

- *See Example 8.13 CommandLineArguments.java*

Sequential Search

- A **Sequential Search** can be used to determine if a specific value (the **search key**) is in an array.
- Approach is to start with the first element and compare each element to the search key:
  - If found, return the index of the element that contains the search key.
  - If not found, return -1.
    - Because -1 is not a valid index, this is a good return value to indicate that the search key was not found in the array.

Code to Perform a Sequential Search

```
public int findWinners( int key )
{
   for ( int i = 0; I<winners.length; i++ )
   {
     if ( winners[i] == key )
           return i;
   }
   return -1;
}
```

- *See Examples 8.14 and 8.15*

Sorting an Array

- When an array's elements are in random order, our Sequential Search method needs to look at every element in the array before discovering that the search key is not in the array. This is inefficient; the larger the array, the more inefficient a Sequential Search becomes.
- We could simplify the search by arranging the elements in numeric order, which is called **sorting the array**. Once the array is sorted, we can use various search algorithms to speed up a search.

Selection Sort

- In a **Selection Sort**,
  - we select the largest element in the array and place it at the end of the array.
  - Then we select the next-largest element and put it in the next-to-last position in the array, and so on.
- To do this, we consider the unsorted portion of the array as a **subarray**.
  - We repeatedly select the largest value in the current subarray and move it to the end of the subarray,
  - then consider a new subarray by eliminating the elements that are in their sorted locations.
  - We continue until the subarray has only one element. At that time, the array is sorted.

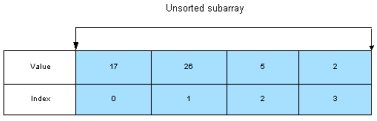
The Selection Sort Algorithm

To sort an array with n elements in ascending order:

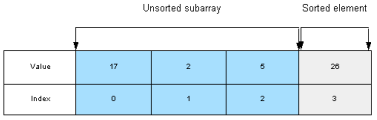
1. Consider the n elements as a subarray with $m = n$ elements.
2. Find the index of the largest value in this subarray.
3. Swap the values of the element with the largest value and the element in the last position in the subarray.
4. Consider a new subarray of $m = m - 1$ elements by eliminating the last element in the previous subarray
5. Repeat steps 2 through 4 until $m = 1$.

Selection Sort Example

- In the beginning, the entire array is the unsorted subarray:

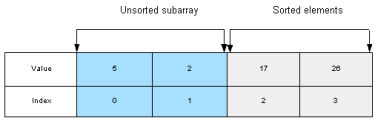


- We swap the largest element with the last element:

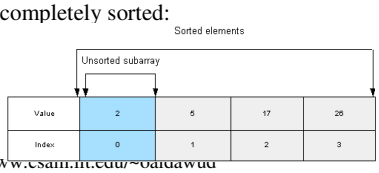


Selection Sort Example (continued)

- Again, we swap the largest and last elements:



- When there is only 1 unsorted element, the array is completely sorted:



Swapping Values

- To swap two values, we define a temporary variable to hold the value of one of the elements, so that we don't lose that value during the swap.

To swap elements *a* and *b*:

1. define a temporary variable, *temp.*
2. assign element *a* to *temp.*
3. assign element *b* to element *a.*
4. assign *temp* to element *b.*

Swapping Example

- This code will swap elements 3 and 6 in an *int* array named *array*:

```
int temp;              // step 1
temp = array[3];       // step 2
array[3] = array[6];   // step 3
array[6] = temp;       // step 4
```

- *See Examples 8.16 & 8.17*

Bubble Sort

- The basic approach to a Bubble Sort is to make multiple passes through the array.
- In each pass, we compare adjacent elements. If any two adjacent elements are out of order, we put them in order by swapping their values.
- At the end of each pass, one more element has "bubbled" up to its correct position.
- We keep making passes through the array until all the elements are in order.

Bubble Sort Algorithm

- To sort an array of *n* elements in ascending order, we use a nested loop:
- The outer loop executes *n – 1* times.
- For each iteration of the outer loop, the inner loop steps through all the unsorted elements of the array and does the following:
  - Compares the current element with the next element in the array.
  - If the next element is smaller, it swaps the two elements.

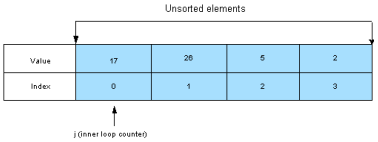
Bubble Sort Pseudocode

```
for i = 0 to last array index – 1 by 1
{
 for j = 0 to ( last array index – i – 1 ) by 1
 {
  if ( 2 consecutive elements are not in order )
    swap the elements
 }
}
```

http://www.csam.iit.edu/~oaldawud

Bubble Sort Example

- At the beginning, the array is:

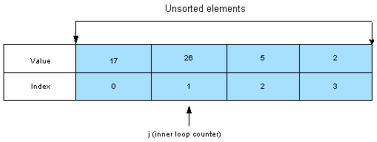


- We compare elements 0 (17) and 1 (26) and find they are in the correct order, so we do not swap.

http://www.csam.iit.edu/~oaldawud

Bubble Sort Example (con't)

- The inner loop counter is incremented to the next element:

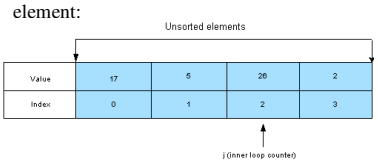


- We compare elements 1 (26) and 2 (5), and find they are not in the correct order, so we swap them.

http://www.csam.iit.edu/~oaldawud

Bubble Sort Example (con't)

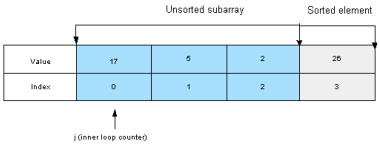
- The inner loop counter is incremented to the next element:



- We compare elements 2 (26) and 3 (2), and find they are not in the correct order, so we swap them.
- The inner loop completes, which ends our first pass through the array.

http://www.csam.iit.edu/~oaldawud

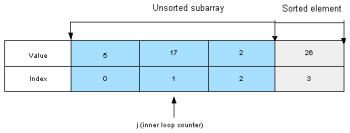
Bubble Sort Example (2nd pass)



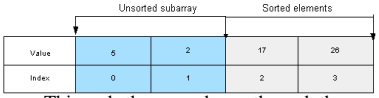
- The largest value in the array (26) has bubbled up to its correct position.
- We begin the second pass through the array. We compare elements 0 (17) and 1 (5) and swap them.

http://www.csam.iit.edu/~oaldawud

Bubble Sort Example (2nd pass)



- We compare elements 1 (17) and 2 (2) and swap.

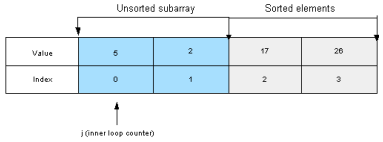


- This ends the second pass through the array. The second-largest element (17) has bubbled up to its correct position.

http://www.csam.iit.edu/~oaldawud

Bubble Sort (3rd pass)

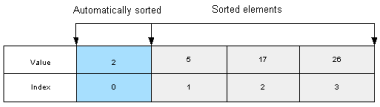
- We begin the last pass through the array.



- We compare element 0 (5) with element 1 (2) and swap them.

Bubble Sort (complete)

- The third-largest value (5) has bubbled up to its correct position.



- Only one element remains, so the array is now sorted.

Bubble Sort Code

```
for ( int i = 0; i < array.length - 1; i++ )
{
  for ( int j = 0; j < array.length - i - 1; j++ )
  {
     if ( array[j] > array[j + 1] )
     {
         // swap the elements
         int temp = array[j + 1];
         array[j + 1] = array[j];
         array[j] = temp;
     }
  } // end inner for loop
} // end outer for loop
```

- *See Examples 8.18 & 8.19*

Sorting Arrays of Objects

- In arrays of objects, the array elements are object references.
- Thus, to sort an array of objects, we need to sort the **data** of the objects.
- Usually, one of the instance variables of the object acts as a **sort key**.
  - For example, in an email object, the sort key might be the date received.

Example

- Code to sort an array of *Auto* objects using *model* as the sort key:

```
for ( int i = 0; i < array.length - 1; i++ )
{
 for ( int j = 0; j < array.length - i - 1; j++ )
 {
    if ( array[j].getModel( ).compareTo(
              array[j + 1].getModel( ) ) > 0 )
    {
      Auto temp = array[j + 1];
      array[j + 1] = array[j];
      array[j] = temp;
    } end if statement
 } // end inner for loop
} // end outer for loop
```


Sequential Search of a Sorted Array

- When the array is sorted, we can implement a more efficient algorithm for a sequential search.
- If the search key is not in the array, we can detect that condition when we find a value that is higher than the search key.
- All elements past that position will be greater than the value of that element, and therefore, greater than the search key.

Sample Code

```
public int searchSortedArray( int key )
{
   for ( int i = 0; i < array.length
                    && array[i] <= key; i++ )
   {
      if ( array[i] == key )
         return i;
   }
   return -1; // end of array reached without
              // finding key or
              // an element larger than
              // the key was found
}
```


Binary Search

- A **Binary Search** is like the "Guess a Number" game.
- To guess a number between **1** and **100**, we start with **50** (halfway between the beginning number and the end number).
- If we learn that the number is greater than **50**, we immediately know the number is not **1 - 49**.
- If we learn that the number is less than **5**0, we immediately know the number is not **51** - **100**.
- We keep guessing the number that is in the middle of the remaining numbers (eliminating half the remaining numbers) until we find the number.

Binary Search

- The "Guess a Number" approach works because **1 - 100** are a "sorted" set of numbers.
- To use a Binary Search, the array must be sorted.
- Our Binary Search will attempt to find a search key in a sorted array.
  - If the search key is found, we return the index of the element with that value.
  - If the search key is not found,we return **-1**.

The Binary Search Algorithm

- We begin by comparing the **middle** element of the array and the search key.
- If they are equal, we found the search key and return the index of the middle element.
- If the middle element's value is greater than the search key, then the search key cannot be found in elements with higher array indexes. So, we continue our search in the left half of the array.
- If the middle element's value is less than the search key, then the search key cannot be found in elements with lower array indexes. So, we continue our search in the right half of the array.

The Binary Search Algorithm (con't)

- As we keep searching, the subarray we search keeps shrinking in size. In fact, the size of the subarray we search is cut in half at every iteration.
- If the search key is not in the array, the subarray we search will eventually become empty. At that point, we know that we will not find our search key in the array, and we return –1.

Example of a Binary Search

- For example, we will search for the value 7 in this sorted array:

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- To begin, we find the index of the center element, which is 8, and we compare our search key (7) with the value 45.

Binary Search Example (con't)

- Because 7 is less than 35, we eliminate all array elements higher than our current middle element and consider elements 0 through 7 the new subarray to search.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 3, so we compare 7 to the value 8.

Binary Search Example (con't)

- Because 7 is less than 8, we eliminate all array elements higher than our current middle element (3) and make elements 0 through 2 the new subarray to search.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 1, so we compare 7 to the value 6.

Binary Search: Finding the search key

- Because 7 is greater than 6, we eliminate array elements lower than our current middle element (1) and make element 2 the new subarray to search.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The value of element 2 matches the search key, so our search is successful and we return the index 2.

Binary Search Example 2

- This time, we search for a value not found in the array, 34. Again, we start with the entire array and find the index of the middle element, which is 8.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- We compare our search key (34) with the value 45.

Binary Search Example 2 (con't)

- Because 34 is less than 45, we eliminate array elements higher than our current middle element and consider elements 0 through 7 the new subarray to search.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 3, so we compare 34 to the value 8.

Binary Search Example 2 (con't)

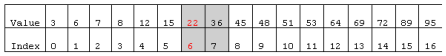
- Because 34 is greater than 8, we eliminate array elements lower than our current middle element and consider elements 4 through 7 the new subarray to search.

Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 5, so we compare 34 to the value 15.

Binary Search Example 2 (con't)

- Again, we eliminate array elements lower than our current middle element and make elements 6 and 7 the new subarray to search.

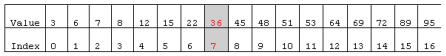


Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- The index of the center element is now 6, so we compare 34 to the value 22.

Binary Search 2: search key is not found

- Next, we eliminate array elements lower than our current middle element and make element 7 the new subarray to search.



Value	3	6	7	8	12	15	22	36	45	48	51	53	64	69	72	89	95
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

- We compare 34 to the value 36, and attempt to eliminate the higher subarray, which leaves an empty subarray.
- We have determined that 32 is not in the array. We return -1 to indicate an unsuccessful search.

Binary Search Code

```
public int binarySearch( int [] array, int key )
{
   int start = 0, end = array.length - 1;

   while ( end >= start )
   {
      int middle = ( start + end ) / 2;
      if ( array[middle] == key )
         return middle;   // key found
      else if ( array[middle] > key )
         end = middle - 1; // search left
      else
         start = middle + 1; // search right
   }
   return -1;  // key not found
}
```


Using Arrays as Counters

- To count multiple items, we can use an array of integers.
- Each array element is a counter.
- Example: we want to throw a die and count the number of times each side is rolled.
  - We set up an array of 6 integer counters, initialized to 0.
  - For each roll, we use (`roll - 1`) as the index of the array element to increment.

Example Code

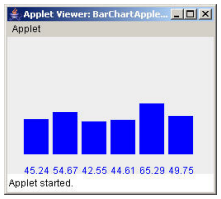
```
// instantiate array of counters
int [] rollCount = new int [6];

// roll the die NUMBER_OF_ROLLS times
for ( int i = 1; i <= NUMBER_OF_ROLLS; i++ )
{
     // roll the die
     int roll = (int) Math.random( ) * 6 + 1;

     // increment the corresponding counter
     rollCount[roll - 1]++;
}
```

- See Examples **8.22, 8.23, & 8.24**

Backup

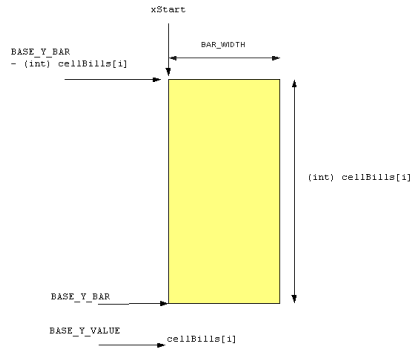
Displaying a Bar Chart

- We can display the data of an array graphically as a bar chart:



- Each bar is drawn as a rectangle using the *llfiRect* method in the *Graphics* class.

Arguments for the *fillRect* Method



Arguments for the *fillRect* Method

- API for the *fillRect* method in the *Graphics* class:

```
void fillRect( int UpperLeftX, int
UpperLeftY,
                int width, int height )
```

- width:
  - the width of the bar is a constant value. For our bar chart, we chose a width of 30 pixels.
- height:
  - the height for each bar is the value of the array element being charted.

Arguments for the *fillRect* Method

- UpperLeftY value:
  - the height of the bar subtracted from the base y value for drawing all the bars. We subtract because y values increase from the top of the window to the bottom.
- UpperLeftX value:
  - the first bar starts at a constant left margin value. After we draw each bar, we position the starting x value for the next bar by incrementing the start x value by the width of the bar, plus the space between bars.

Drawing the Bar Chart

```
int xStart = LEFT_MARGIN;  // first bar
for ( int i = 0; i < cellBills.length; i++ )
{
   g.fillRect( xStart,
            BASE_Y_BAR - (int)( cellBills[i]
  ),
            BAR_WIDTH, (int)( cellBills[i] )
  );
   g.drawString( Double.toString(
  cellBills[i] ),
            xStart, BASE_Y_VALUE );


   // move to starting x value for next bar

   xStart += BAR_WIDTH + SPACE_BETWEEN_BARS;
}
```