COP 3330: Object-Oriented Programming Summer 2007

Arrays and Strings in Java – Part 1

Instructor : Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2007

School of Electrical Engineering and Computer Science
University of Central Florida

Strings

- A ***string*** is a sequence of characters.
- Java provides two classes to support strings:
  - String class – the instances of String class are constants, i.e., the content of a String object cannot be changed after its creation.
  - StringBuffer class – the instances of StringBuffer class are mutable, i.e., the content of a StringBuffer object can be changed after its creation.
- The String class has some unique privileges not shared by ordinary classes.
  - A string can be created using string literals.
  - Operator + can be applicable to strings.
- The characters in a string are indexed by 0 to n-1, where n is the length of the string.

`String` Class

- Creation of a String object.

```
String s = new String("abc");
String s = "abc";
```

- Add operator:

```
String s1 = "abc";
String s2 = "de";
String s3 = s1 + s2; //s3 = "abcde"
```

- The `String` class has a lot of methods. These methods are useful to manipulate strings.

`length` and `charAt` methods

`int length()` – this instance method returns the length of the string.

```
String s="abc";
s.length()    ➔ returns 3
```

`char charAt(int index)` – this instance method returns the character at a specific index.

```
String s="abcde";
s.length();    ➔ 5
s.charAt(0);   ➔ 'a'
s.charAt(1);   ➔ 'b'
s.charAt(4);   ➔ 'e'
s.charAt(5);   ➔ error
```


`indexOf` method

```
int indexOf(char c)
int indexOf(char c, int index)
```

Returns the index of the first occurrence of the character c in the current object starting at position `index` (default 0). Returns –1 if there is no such occurrence. [overloaded method]

```
String s="ababc";
s.indexOf('b');//returns 1
s.indexOf('b',2);   //returns 3
s.indexOf('d',2); //returns -1
```


`indexOf` method (cont.)

`int indexOf(String s2)`

`int indexOf(String s2, int index)`

Returns the index of the first occurrence of the substring s2 in the current object, beginning at position `index` (default 0). Returns –1 if there is no such occurrence.

```
String s="daabcabc";
String s2="ab";
s.indexOf(s2);   //returns 2
s.indexOf(s2,3); //returns 5
s.indexOf(s2,6); //returns -1
```


`substring` method

`String substring(int startindex)`
`String substring(int startindex, int lastindex)`

Returns the substring of the current object starting from `startindex` and ending with `lastindex-1` (or the last index of the string if `lastindex` is not given). [overloaded method]

```
String s="abcdef";
s.substring(1);  //returns "bcdef"
s.substring(3);  //returns "def"
s.substring(1,4);//returns "bcd"
s.substring(3,5);//returns "de"
```


`equals` and `equalsIgnoreCase`

`boolean equals(String s2)`
`boolean equalsIgnoreCase(String s2)`

Returns true if the current object is equal to s2; otherwise false.

The method `equalsIgnorecase` disregards the case of the characters.

```
String s="abc";
s.equals("abc")   //returns true
s.equals("abcd")  //returns false
s.equals("aBc")   //returns false
s.equalsIgnoreCase("aBc")//returns true
```


`StringBuffer` Class

- `StringBuffer` constructors:

  `StringBuffer()`
  `StringBuffer(int size)`

  Returns an instance of the StringBuffer class that is empty but has an initial capacity of `size` characters (default 16 characters).

  `StringBuffer(String arg)`

  Creates an instance of the StringBuffer class from the string arg.

- `length` and `charAt` methods are also defined for StringBuffer class.

`append` and `insert` methods

`StringBuffer append(String s)`

Returns the current object with the String parameter s appended to the end.

`StringBuffer insert(int index, char c)`

`StringBuffer insert(int index, String s)`

Inserts the character c (or String s) into the current StringBuffer object at index `index`. The characters (after `index`) are shifted to right.

```
sb = new StringBuffer("abcd");
sb.insert(0,"AB") //returns sb as "ABabcd"
sb.insert(1,"CD") //returns sb as "ACDBabcd"
sb.insert(8,"EFG")//returns sb as "ACDBabcdEFG"
sb.append("HI")   //returns sb as "ACDBabcdEFGHI"
```


Reading strings using `BufferedReader`

- Recall from our earlier look at using `BufferedReader` that we need to first declare a `BufferedReader` object that was directed to read from the input stream attached to the standard input device. Once this is done, reading strings from the keyboard is straightforward.

```
BufferedReader stdin = new BufferedReader(
        new InputStreamReader(System.in));
```


BufferedReader Example

```
//Developer: Mark Llewellyn       Date: June 2007
import java.io.*;

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

      BufferedReader stdin = new BufferedReader(new
        InputStreamReader(System.in));
      System.out.println("Enter your first name: ");
      String firstName = stdin.readLine();
      System.out.println("Enter your last name: ");
      String lastName = stdin.readLine();
      System.out.println("Your name is " + firstName + " "
        + lastName + ".");
    }
}
```


Wordlength Example

```
//Developer:  Mark Llewellyn  Date: June 2007
import java.io.*;

public class Wordlength {
   public static void main(String[] args) throws IOException {
     BufferedReader stdin = new BufferedReader(
        new InputStreamReader(System.in));

   //read the word from the user
   System.out.println("Enter a word: ");
   String word = stdin.readLine();

   //determine the length of the word
   int wordLength = word.length();

   //output results
   System.out.println("Word " + word + " has a length of "
        + wordLength + " characters.");
   }
}
```


Palindrome Example

```
//Developer: Mark Llewellyn       Date: June 2007
//Checks words to see if they are palindromes
import java.io.*;

public class Palindrome {
 static boolean isPalindrome(String s) {
    int i = 0;
    int j = s.length() - 1;
    boolean flag = true;
    while ((i<j) && flag){
       if (s.charAt(i) != s.charAt(j))
          flag = false;
       else {i++; j--; }
    }
    return flag;
 }
```


Palindrome Example (cont.)

```
public static void main(String args[]) throws
   IOException {
     String s;
     BufferedReader stdin =
        new BufferedReader (new InputStreamReader
   (System.in));

     System.out.print("A String > ");
     System.out.flush();
     s = stdin.readLine();

     if (isPalindrome(s))
        System.out.println(s + " is a palindrome");
     else
        System.out.println(s + " is not a palindrome");
 }
}
```


Decimal to Binary -- Example

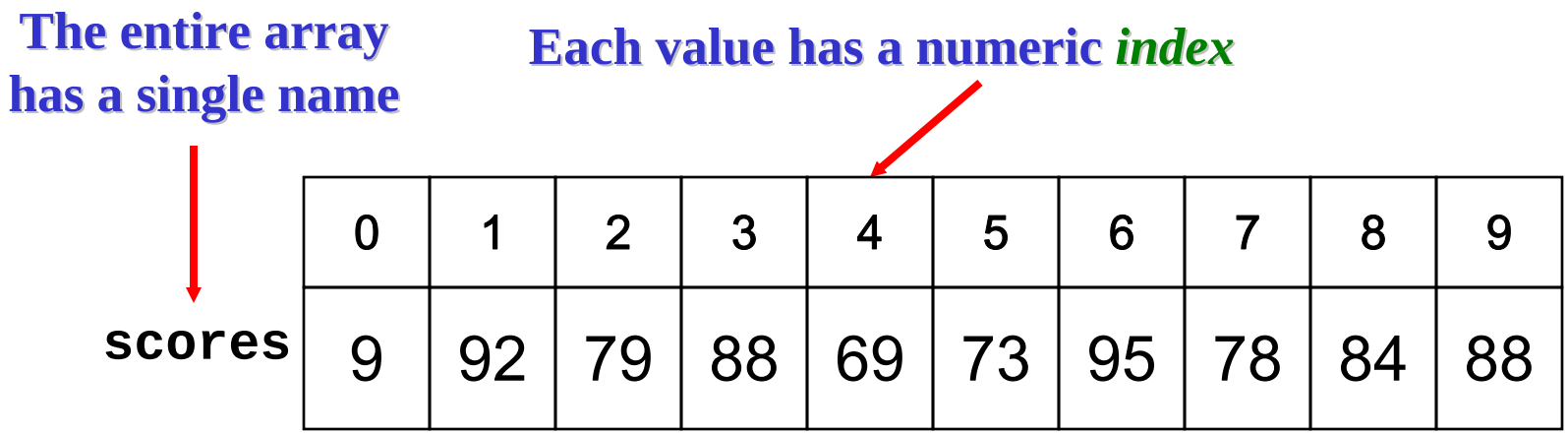
```
//Developer: Mark Llewellyn       Date: June 2007
//Converts a decimal number to its binary equivalent
import java.io.*;
public class DecToBinary {
  static StringBuffer toBinary(int decVal) {
    StringBuffer sb = new StringBuffer("");
    if (decVal == 0)
       sb.insert(0,"0");   //note insert position
    else
       while (decVal != 0) {
          if (decVal%2 == 0)
             sb.insert(0,"0"); //note insert position
          else
             sb.insert(0,"1");  //note insert position
          decVal = decVal / 2;
       }
    return sb;
 }
```


Decimal to Binary – Example (cont.)

```
public static void main(String args[]) throws IOException {
    int num;
    BufferedReader stdin =
       new BufferedReader (new InputStreamReader(System.in));

    System.out.print("An integer > ");
    System.out.flush();
    num = Integer.parseInt(stdin.readLine().trim());

    System.out.println("Decimal Number: " + num +
       "   Corresponding Binary Number: "  + toBinary(num));
 }
}
```


Arrays in Java

- **An *array* is an ordered list of values, each of the same type.**



This array holds 10 values that are indexed from 0 to 9

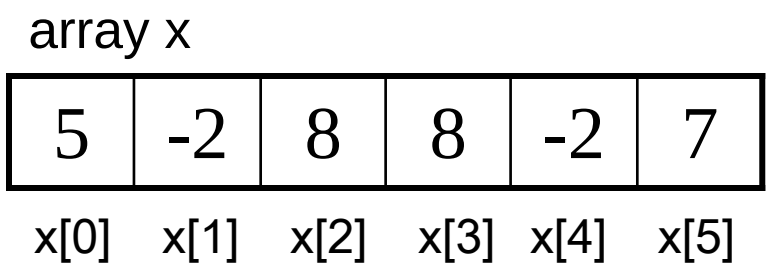
An array of size N is indexed from zero to N-1

An array type variable holds a reference to an object.

Accessing Elements in Arrays

- The act referring to an individual array element is called *subscripting* or *indexing*.
  - general form: **`arrayname[index]`**

    where `index` is an integer expression.



Examples

```
int[] x = new int[6];
x[0]=5;
i=2;
x[i+1]=x[i];
```


Explicit Initialization of Arrays

- Java provides a declaration form to explicitly specify the initial values of the elements of an array.
- Assuming an n element array, the general form is:

```
ElementType[] id = {expr0, expr1, ..., exprn-1};
```

where each $expr_i$ is an expression that evaluates to type `ElementType`.

Examples:

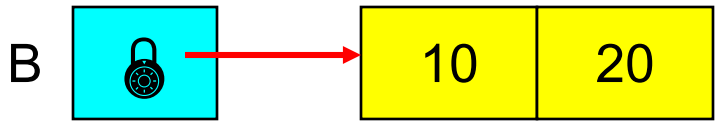
```
int[] fibonacci = {1, 1, 2, 3, 5, 8, 13, 21, 34};
String[] cats = {"bug", "squeaky", "paris", "kitty kat"};
int[] unit = {1};
```


Constant Arrays

- As in other declarations, the modifier final can be applied in an array declaration. The modifier has the usual effect – after its initialization, the array variable is treated as a constant. In this case, the array variable cannot change which array it references, however, the values in the array can be changed!

```
final int[] B = {10, 20};
```

- Graphically this would look like the following:



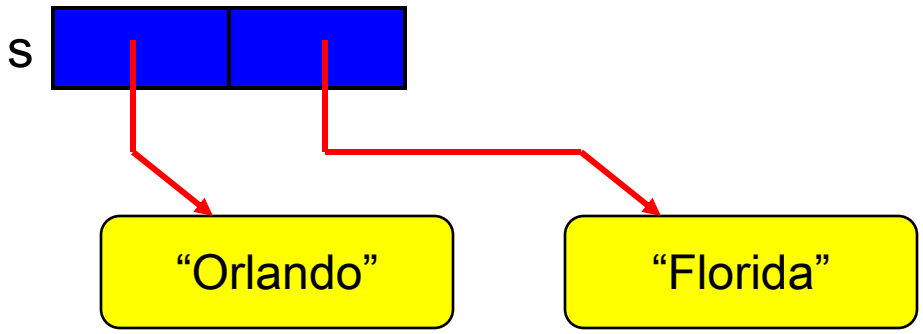
where 🔒 indicates that the value of the reference cannot be changed.

```
B = new int[2]; //illegal: a final cannot be the
                //target of an assignment

B[1] = 30;      //legal: B[1] is not final
```


Arrays of Objects

- If the element type of an array is an object type, then the elements hold either the value null or references to element type values rather than holding element type values themselves. This is illustrated graphically below:

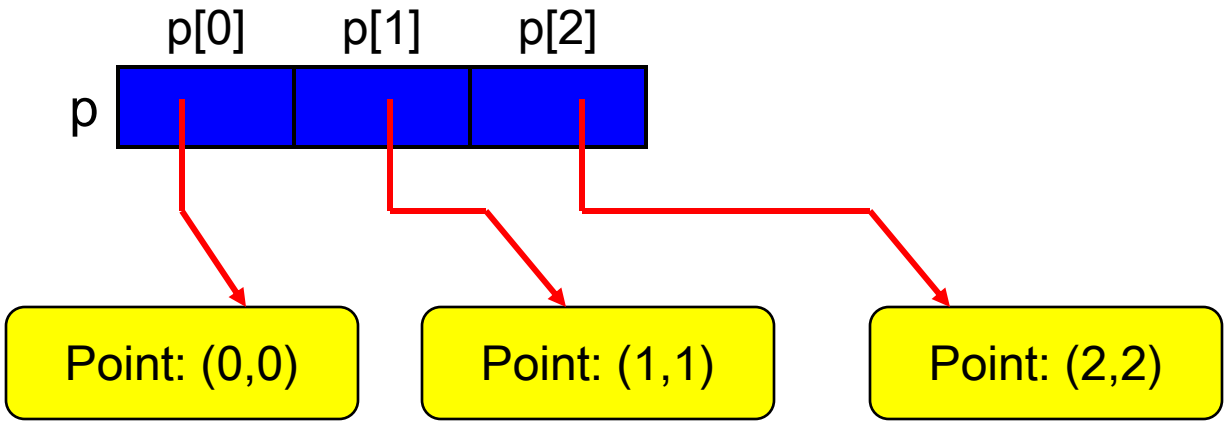
```
String[] s = new String[2];
s[0] = "Orlando";
s[1] = "Florida";
```



Arrays of Objects (cont.)

- This example uses the `awt` class `Point`.

```
Point[] p = new Point[3];
p[0] = new Point(0,0);
p[1] = new Point(1,1);
p[2] = new Point(2,2);
```

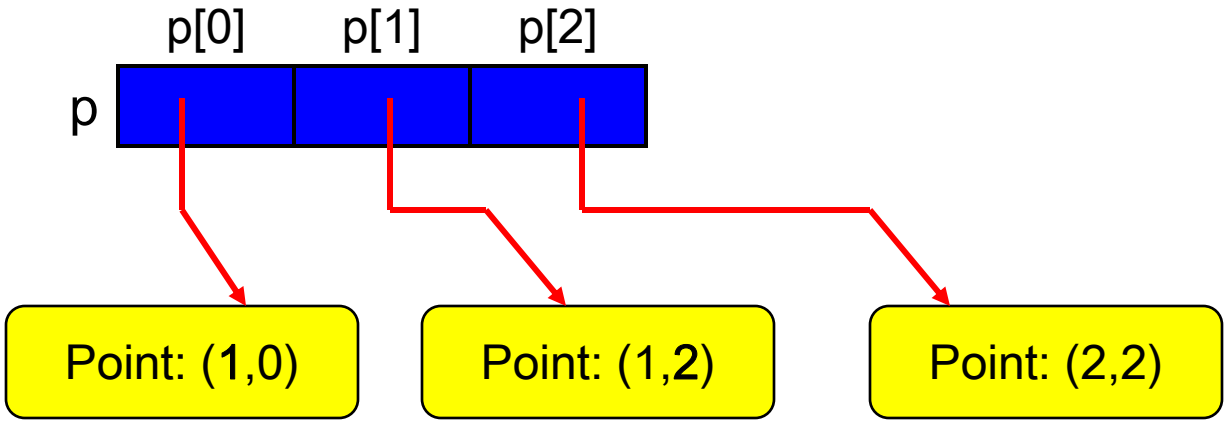


Arrays of Objects (cont.)

- Because `p[0]`, `p[1]`, and `p[2]` are `Point` reference variables, they have access to `Point` member methods. For example the code segment below:

```
p[0].setX(1);
p[1].setY(p[2].getY());
```

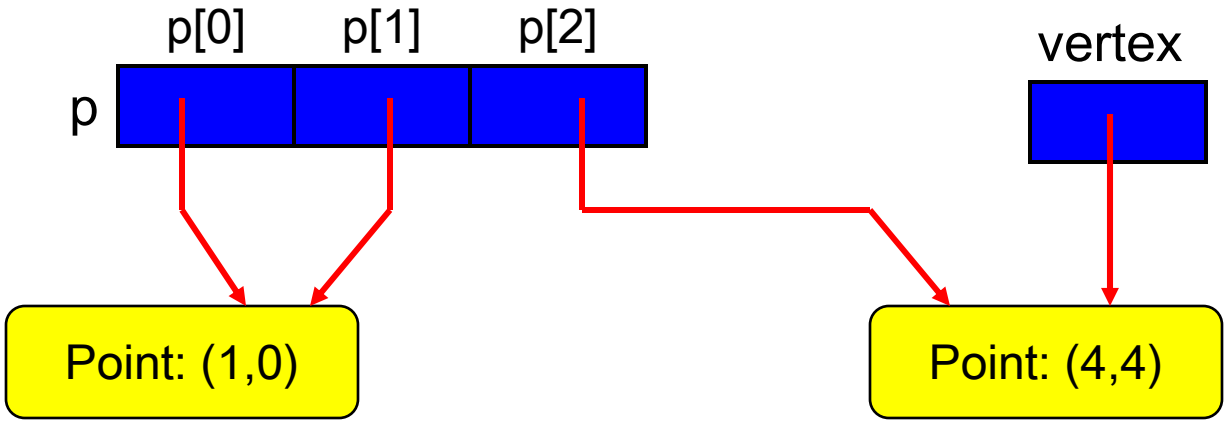
Causes the following update of array `p`.



Arrays of Objects (cont.)

- The elements of p hold references. Therefore, an assignment of one of its elements modifies what that element references rather than modifying the object that had been referenced. Consider the following code segment that further updates array p.

```
Point vertex = new Point(4,4);
p[1] = p[0];
p[2] = vertex;
```



Array Bounds in Java

- Unlike many programming languages, Java automatically checks that proper array subscripts are used. If a subscript is determined to be invalid, an exception of type `IndexOutOfBoundsException` is generated. (Unless the program provides exception handling code, this exception causes the program to terminate.)
- The following code segment contains two misuses of array `sample`.

```
int[] sample = new int[40];
sample[-1] = 0; //subscript too small
sample[40] = 10; //subscript too large
```


Arrays in Java (cont.)

- Arrays can be defined with or without initialization. If there is no initialization, then the definition has the form:

```
ElementType[]  id;
```

where `ElementType` is the type of the individual elements in the array and `id` is an identifier naming the array. The value of any location (index) in this array is currently undefined.

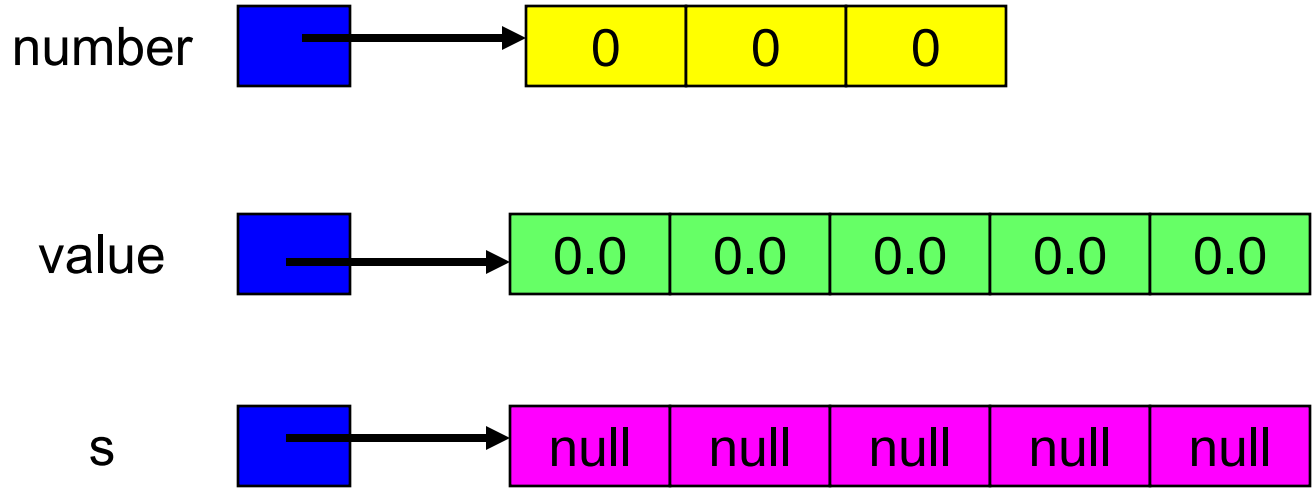
- An array variable definition can specify the array to which the variable is to reference. The most common form is:

```
ElementType[] id = new ElementType[n];
```

where n is a nonnegative integer expression specifying the number of elements in the array.

Array Definition - Example

```
BufferedReader stdin = new BufferedReader(
   new InputStreamReader(System.in));
int[] number = new int[3];
n = Integer.parseInt(stdin.readLine()); //assume n = 5
double[] value = new double[n];
String[] s = new String[n];
```



Array Definition - Example

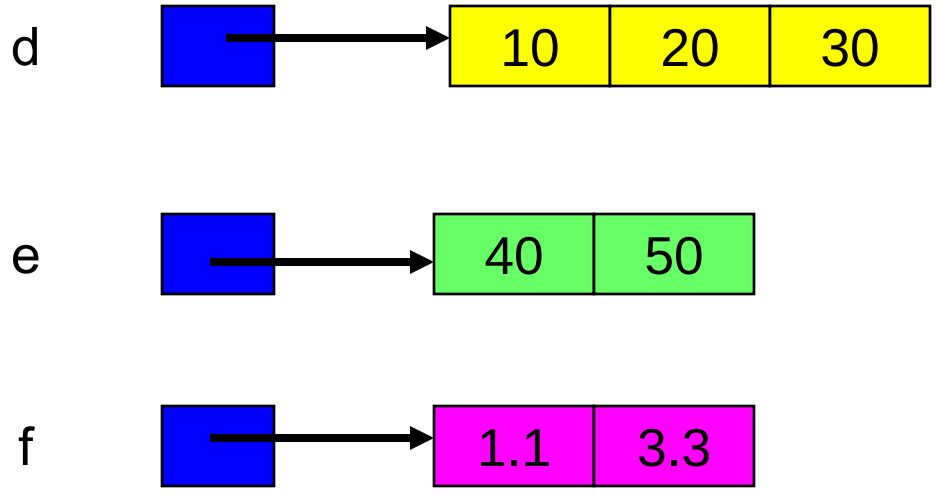
- Since no initial values were specified for the elements themselves, the elements in all three arrays are default initialized.

By default:

- **numeric arrays are initialized to 0.**
- **boolean arrays are initialized to false.**
- **object arrays are initialized to null.**

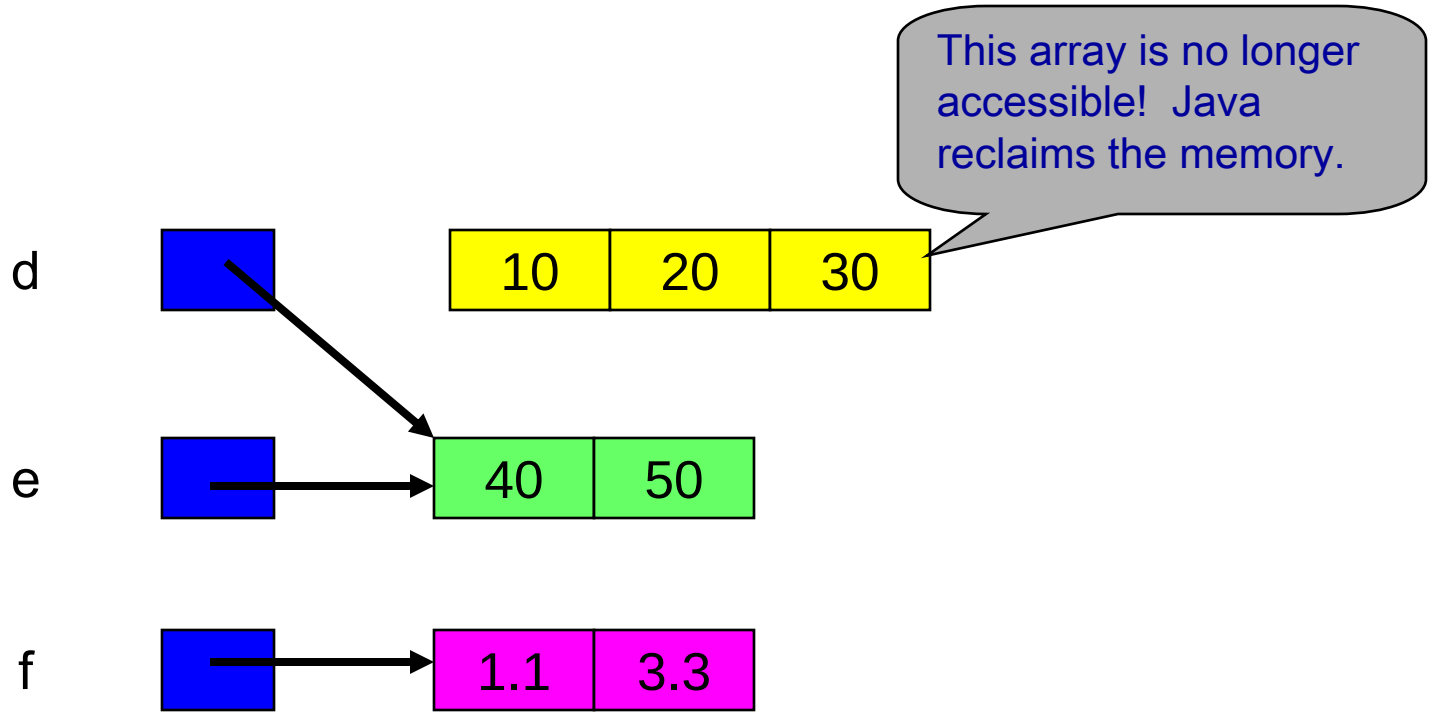
Arrays in Java (cont.)

- Suppose `d` and `e` are both `int` arrays, where `d` references an array with three elements whose values are initialized to 10, 20, and 30 and `e` references an array of two elements whose values are initialized to 40 and 50. Also suppose that `f` is a two-element `double` array whose elements are initialized to 1.1 and 3.3. This scenario is shown below:



Arrays in Java (cont.)

- Since `d` and `e` are both of the same type (`int` arrays), we can assign one to the other For example, if `d = e` is executed. Then the new value of `d` is the value of `e`; i.e., `d` and `e` now reference the same array of elements. Graphically, this will look like the figure below:



Member Methods and Arrays in Java

- Because an array is a Java object, an array as associated with it all of the member methods of an `Object` (e.g., `clone()` and `equals()`). However, the array `clone()` method is overridden specifically for arrays.
- In addition to member methods, an array also has a **`public final`** data field `length` specifying the number of elements in the array. Designers of Java were quite helpful to programmer by including this field as it is a convenience not found in many other programming languages. This allows the programmer to do the following:

```
int fibonacci = {1, 1, 2, 3, 5, 7, 13, 21, 34, 55};
for (int i = 0; i < fibonacci.length; ++i){
        System.out.println("Fibonacci Number " + i +
                " is: ", fibonacci[i]);
}
```

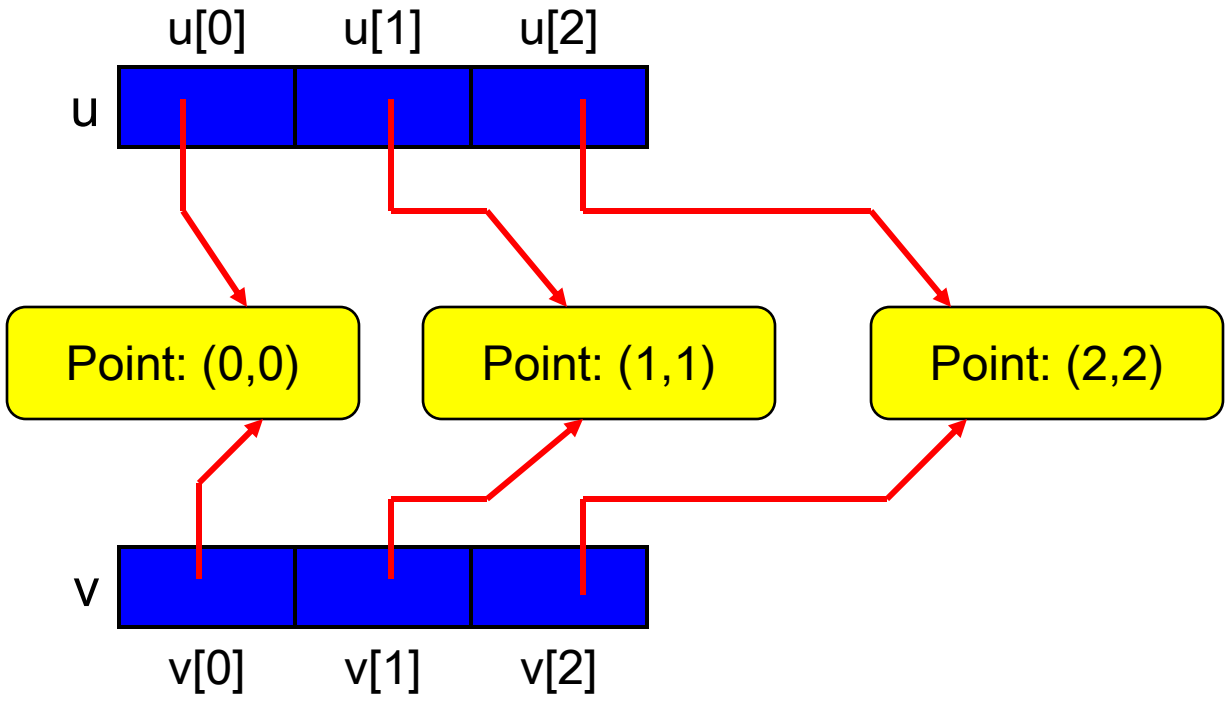

Cloning an Array in Java

- Array method clone() returns a duplicate of the array. The new array object has the same number of elements as the invoking array. The values of the clone array are duplicates of the invoking array.
- A clone produced in this manner is known as a **shallow copy**. The corresponding elements in the two arrays reference the same objects. The shallowness can be seen by viewing the representation of the array variables u and v defined in the following code segment and shown graphically on the next page.

```
Point[] u = {new Point(0,0), new Point(1,1),
             new Point(2,2) };
Point[] v = u.clone();
```

Cloning an Array in Java (cont.)

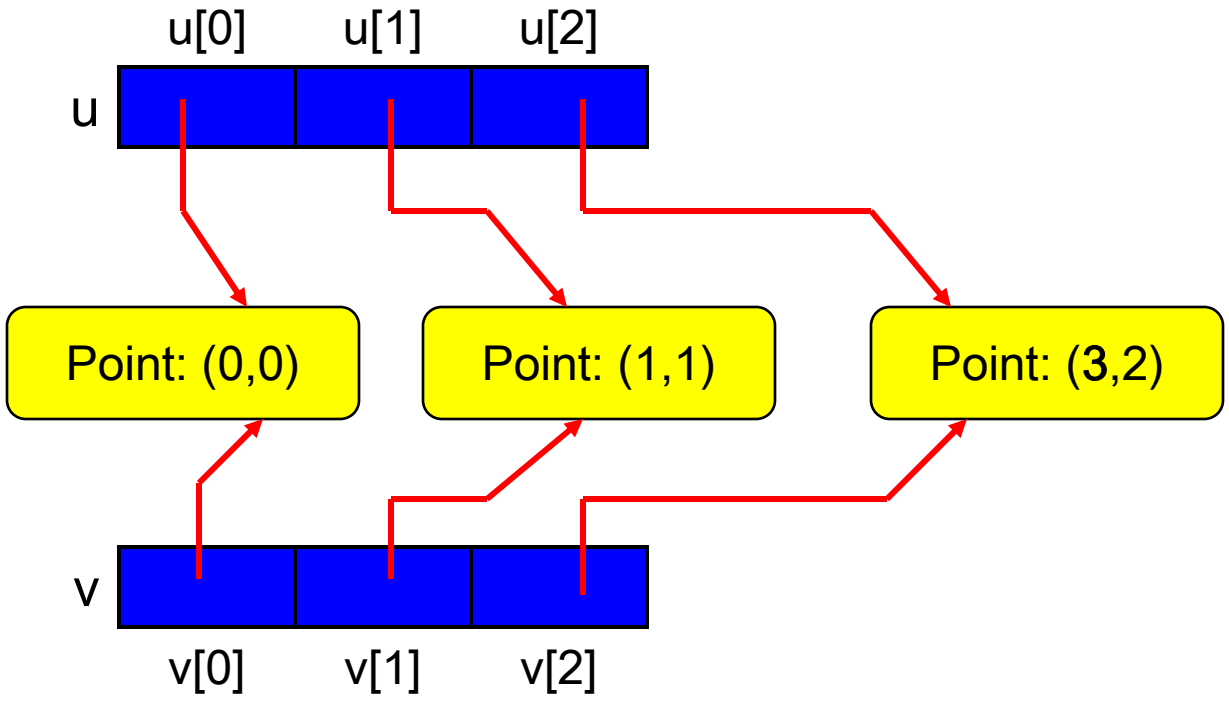
```
Point[] u = {new Point(0,0), new Point(1,1),
                 new Point(2,2) };
Point[] v = u.clone();
```



Cloning an Array in Java (cont.)

- The shallowness of array cloning is further highlighted by the effect of the following statement on the arrays u and v:

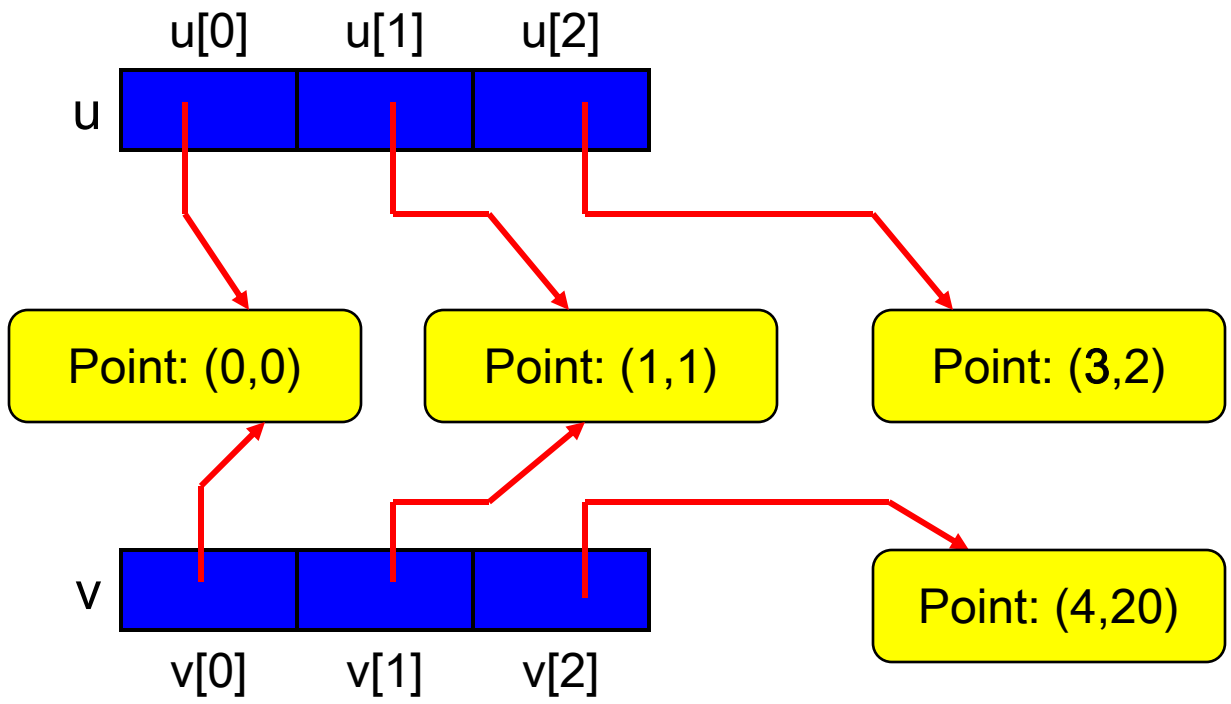
```
u[2].setX(3);
```



Cloning an Array in Java (cont.)

- Although array `v` was created to be a clone of array `u`, there is no requirement that the elements of `v` reference the same `Point` object as `u`.

```
v[2] = new Point(4,20);
```



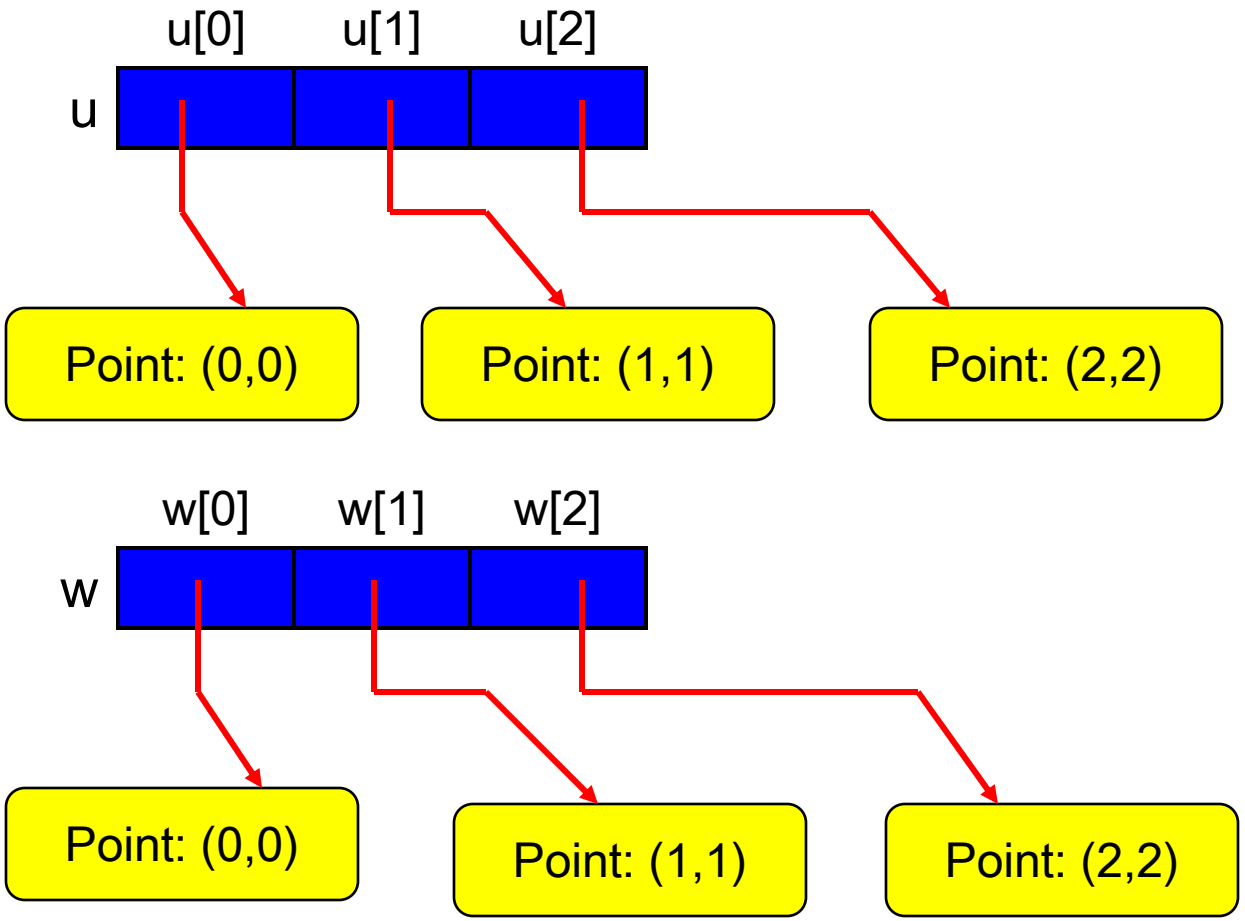
Distinct Element by Element Cloning

- If a distinct element-by-element clone, say `w`, of `u` is needed, it can be created in the following manner.

```
Point[] w = new Point[u.length];
for (int i = 0; i < u.length; ++i) {
      w[i] = u[i].clone();
}
```

- Array clones created in this fashion are called **deep copies**.
- The result of executing the code shown above is illustrated on the next page.

Distinct Element by Element Cloning

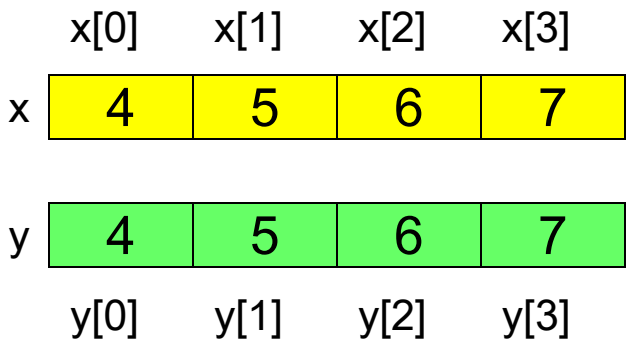


Cloning an Array of a Primitive Type

- Using the method clone() on an array with a primitive element type (not an object type) automatically produces a deep copy.
- The code shown below illustrates this concept.

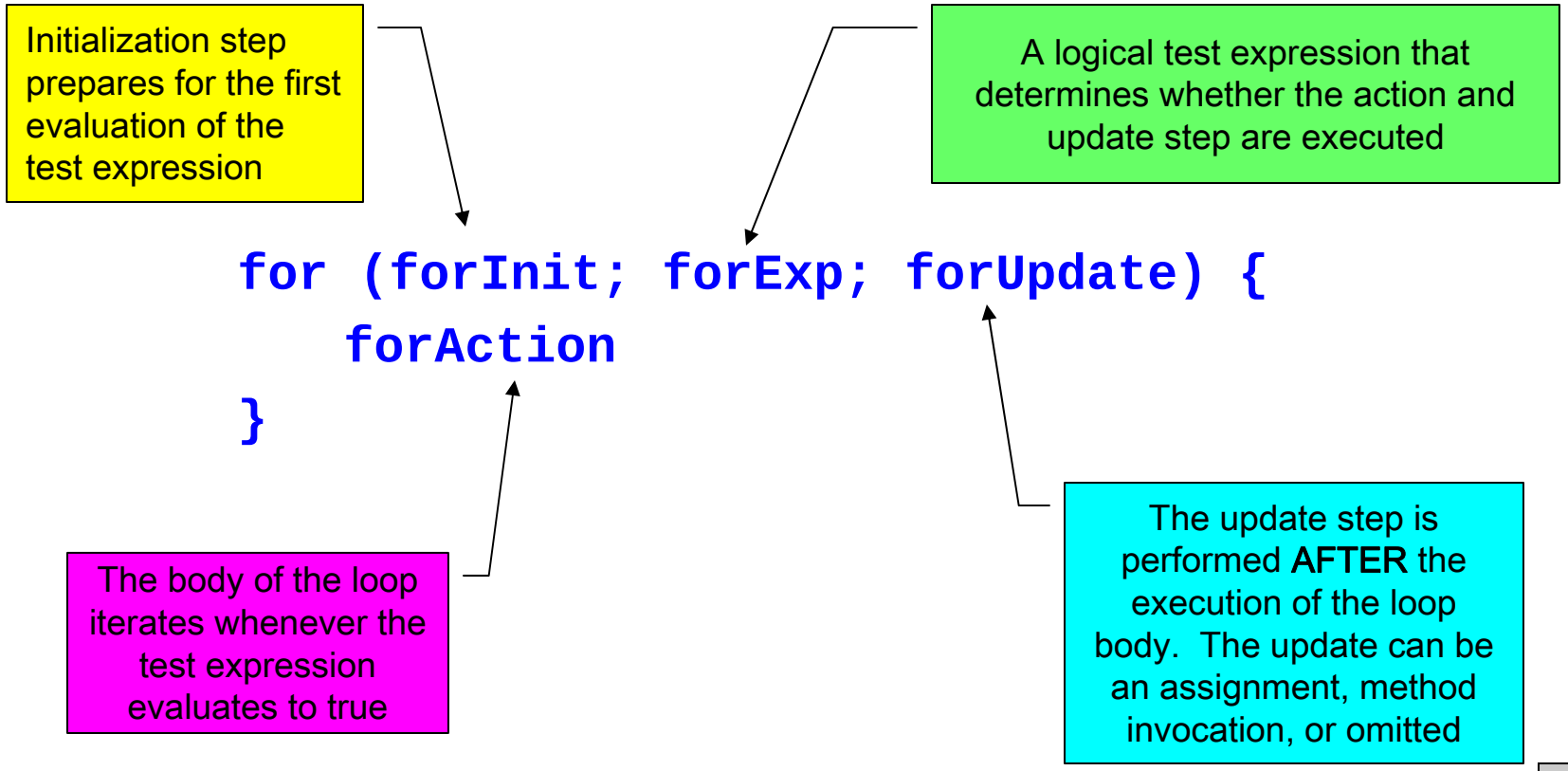
```
int[] x = {4, 5, 6, 7};
int[] y = x.clone();
```

- These two arrays have the following depiction:



For Statement Review

- The next couple of pages are intended as a review and clarification of the `for` statement.



For Statement Revisited Example

Initialization step:
i is set to 0

logical test:
1st iteration: true 0 < 3
2nd iteration: true 1 < 3
3rd iteration: true 2 < 3
4th iteration: false 3 !< 3

```
int n = 3;
for (int i = 0; i < n; i++) {
    System.out.println("Hi!");
}
```

Loop body:
1st iteration: print "Hi!"
2nd iteration: print "Hi!"
3rd iteration: print "Hi!"
4th iteration: not executed

Update step:
After 1st iteration: i = 1
After 2nd iteration: i = 2
After 3rd iteration: i = 3
Not executed 4th time

For Loop Examples

- Note that the loop update is performed only after the body of the loop is executed.
- What do you expect to be printed by the following code segment?

```
int n = 1;
int i;
for (i = 1; i < n; ++i) {
        n++;
}
System.out.println("Value of i is: " + i);
System.out.println("Value of n is: " + n);
```

Output:

```
Value of i is: 1
Value of n is: 1
```

WHY?

For Loop Examples (cont.)

WHY?

– Because the test expression evaluates to false the very first time since `i == 1` and `n == 1`. Since the loop body is never executed, the loop increment is never executed either and thus, the value of `i` remains at `1` when the next statement after the loop is executed.

```
int n = 1;
int i;
for (i = 1; i < n; ++i) {
       n++;
}
System.out.println("Value of i is: " + i);
System.out.println("Value of n is: " + n);
```


For Loop Examples (cont.)

- Will the two loops below generate the same values of `i` at the end of their execution? Yes or No?

```
int n = 5;
int i;
for (i = 1; i < n; ++i){
    System.out.println("Loop 1 – i is: " + i);
}
System.out.println("Final value of i is: " + i);
for (i = 1; i < n; i++){
    System.out.println("Loop 2 - i is: " + i);
}
System.out.println("Final value of i is: " + i);
```

WHY?

For Loop Examples (cont.)

Answer: Yes!

```
Loop 1 – i is: 1
Loop 1 – i is: 2
Loop 1 – i is: 3
Loop 1 – i is: 4
Final value of i is: 5
Loop 2 – i is: 1
Loop 2 – i is: 2
Loop 2 – i is: 3
Loop 2 – i is: 4
Final value of i is: 5
```

WHY?

– The final iteration of the loop occurs in both loops when `i` has a value of `4`. Immediately after this, the loop increment is evaluated which sets `i = 5`. Next the loop expression is evaluated again but this time it is false since `i` is not less than `n` (it is in fact equal to `n` now). The loop terminates in both cases with `i` equal to `5.`

For Loop Increment Step

- Notice in the previous example that it does not matter whether the loop increment variable is updated using the prefix increment operator or the postfix increment operator.
- Since the loop increment occurs only *after* the execution of the loop

Array -- Examples

Reading values of an `int` array:

```
int[] x = new int[100];
for (i=0; i<x.length; i++)
    System.out.println("Enter an integer: ");
    x[i] = Integer.parseInt(stdin.readLine().trim());
}
```

Shifting values of an `int` array:

```
int temp = x[0];
for (i=0; i<x.length; i++)
    x[i]=x[i+1];
x[x.length-1]=temp;
```


Array – Examples (cont.)

Finding the location of the maximum value in an `int` array:

```
int loc = 0;
for (i=1; i<x.length; i++)
  if(x[i]>x[loc])
    loc=i;
```

Finding the first location of a given value in an `int` array:

```
int loc, i=0;
while ((i<x.length) && (x[i]!=item))
    i=i+1;
if (i>=x.length) loc=-1;// not found
else loc=i;             // found
```


Array – Examples (cont.)

Finding summation of the values in an `int` array:

```
int sum=0;
for (i=0; i<x.length; i++)
    sum=sum+x[i];
```

Reversing the contents of an `int` array:

```
int temp,i,j;
for (i=0,j=x.length-1; i<j; i++, j--) {
    temp=x[i];
    x[i]=x[j];
    x[j]=temp;
}
```


Array – Examples (cont.)

Count the number of items in an array which are less than 0.

```
int count;  // For counting the items.
count = 0;  // Start with 0 items counted.
for (int i = 0; i < A.length; i++) {
    if (A[i] < 0.0) // if this item is less than
   zero...
          count++;   // ...then count it
}
// At this point, the value of count is the number
// of items that have passed the test of being < 0
```


Array – Examples (cont.)

- A more complete example of reading values into an array. Similar to the example on page 47.

```
final int MAX_LIST_SIZE = 1000;
int[] buffer = new int[MAX_SIZE_LIST];
int listSize = 0;
for (int i = 0; i < MAX_LIST_SIZE; ++i) {
    String input = stdin.readLine();
    if(input != null) {
            int number = Integer.parseInt(input);
            buffer[i] = number;
            ++listSize;
    }
    else {
            break;
    }
}
int[] date = new int[listSize];
for(i = 0; i < listSize; ++i){
  data[i] = buffer[i];
}
```


Bounds Checking

- Once an array is created, it has a fixed size.
- An index used in an array reference must specify a valid element.
- That is, the index value must be in bounds (0 to N-1).
- The Java interpreter will throw an exception if an array index is out of bounds .
- This is called automatic *bounds checking*.

Bounds Checking

- For example, if the array `codes` can hold 100 values, it can only be indexed using the numbers 0 to 99.
- If `count` has the value 100, then the following reference will cause an `ArrayOutOfBoundsException`:

```
System.out.println (codes[count]);
```

- It's common to introduce *off-by-one errors* when using arrays.

problem

```
for (int index=0; index <= 100; index++)
   codes[index] = index*50 + epsilon;
```

Copying Arrays

To make a new array that is a copy of **A**, it is necessary to make a new array object and to copy each of the individual items from A into the new array:

```
// Make a new array object, the same size as A.
double[] B = new double[A.length];
for (int i = 0; i < A.length; i++)
   B[i] = A[i];   // Copy each item from A to B
```

To make a copy of our sample array **A**, it is **not** sufficient to say:

```
double[] B = A;
```

It does not create a new array object!

All it does is declare a new array variable **B** and make it refer to the same object to which **A** refers.

Copying Arrays (cont.)

- Java has a predefined subroutine to copy values from one array to another . It is static member of the standard System class.
  `System.arraycopy()`
- Its declaration has the form:

```
public static void arraycopy(Object sourceArray,
                             int sourceStartIndex,
                             Object destArray,
                             int destStartIndex,
                             int count)
```


Copying Arrays (cont.)

- **`sourceArray, destArray`** can be arrays with any base type.
- **`count`** tells how many elements to copy.

- Values are taken from **`sourceArray`** starting at position **`sourceStartIndex`** and are stored in **`destArray`** starting at position **`destStartIndex.`**

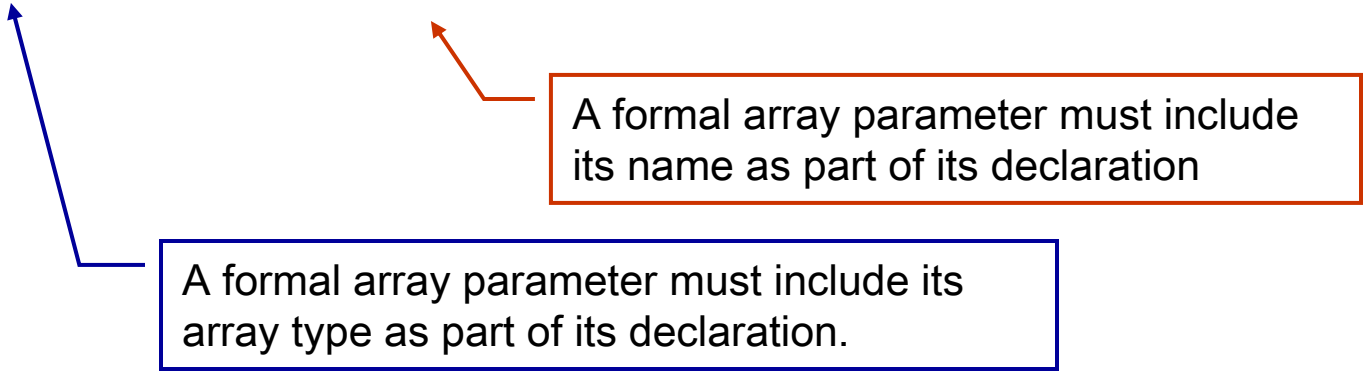
- For example, to make a copy of the array **`A`**:

```
double B = new double[A.length];
System.arraycopy( A, 0, B, 0, A.length );
```


Passing Arrays to Methods

- The syntax for expressing an array parameter definition is no different from that of a normal variable definition. An array parameter has the form:

```
ElementType[] ParameterName
```



- As required by Java, the main method of a console application program has a single array parameter with an element type of String. These are used for command line arguments and we'll look at this more closely later.

Passing Arrays to Methods

- Since arrays are also objects, they are passed into method by call-by-reference.
- A pointer to the array is passed into the method.
- The contents of the array pointed to by an actual parameter can be changed by the method.

```
static void m(int[] x) {
  int i, temp;
  temp=x[0];
  for (i=0; i<(x.length-1); i++)
    x[i]=x[i+1];
  x[x.length-1]=temp;
}
// in main
int[] a = {3,5,7,8};
m(a);      // the content of a will be changed by m
```


Array Copy - Example

A method that makes a copy of an array of doubles:

return type *formal parameter type*

```
double[]  copy( double[] source ) {
// Create and return a copy of the array, source.
// If source is null, return null.
   if ( source == null )
       return null;
   double[]  cpy;  // A copy of the source array.
   cpy = new double[source.length];
   System.arraycopy( source, 0, cpy, 0, source.length );
   return cpy;
}
```


Passing Arrays into Methods -- Example

```
static void findprimes(int[] p, int n) {
  int i, val;
  p[0]=2;  val=3;  i=1;
  while (i<n) {
    if (isprime(val,p,i)) { p[i]=val;  i++; }
    val=val+1;
  }
}

static boolean isprime(int v, int[] p, int lastprimeloc) {
  boolean primeflag=true;
  int i=0;
  while (primeflag && (i<lastprimeloc))
    if (v%p[i] == 0) {primeflag=false;}
    {else i++;}
  return primeflag;
}
```


Passing Arrays into Methods – Example (cont.)

```
// in main
int i;
int[] primes = new int[100];

findprimes(primes,10);
System.out.println("First 10 Primes:");
for (i=0; i<10; i++)
   System.out.println(primes[i]);
```


Sequential Searching in an Array

- Method `sequentialSearch()` is similar to the code on page 48. However, this time we are developing a complete method which we will be able to put into a package of array handling methods called **`ArrayTools`**. Notice that this method is a class (i.e. static) method.

```
// sequentialSearch():  searches unsorted list for a key
public static int sequentialSearch(int[] data, int key){
   for(int i = 0; i < data.length; i++) {
       if (data[i] == key){
             return(i);  //return position of key value
       }
   }
   return –1;
}
```


Sequential Searching in an Array (cont.)

	0	1	2	3	4	5	6	7	8	9	10
A	6	9	82	69	16	54	90	44	2	87	74

```
int[] A = {6, 9, 82, 69, 16, 54,90, 44, 2, 87, 74};
int i1 = ArrayTools.sequentialSearch(A,16);  //4
int i2 = ArrayTools.sequentialSearch(A,45); //-1
System.out.println(Math.max(A[0], A[8]); //6
```


Searching an Ordered Array

- When the values of an array are in sorted order, there are more efficient searches than sequential searching for finding a particular key value. Method `binarySearch()` conducts a series of tests using a divide and conquer strategy to continually reduce the "size" of the array being searched until the position in which the element must be located is found.

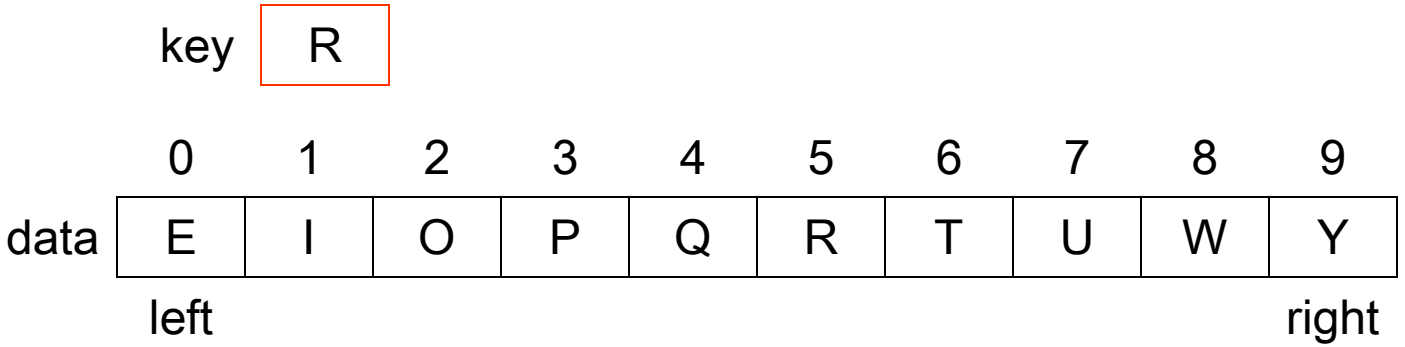
- We will include this method in our package **`ArrayTools`**. Notice that this method is also a class method.

- NOTE: The standard Java class Collections method binarySearch() behaves somewhat differently than the method on the next page – if the key value is not in the list the built-in method returns the index where the key value should be located.

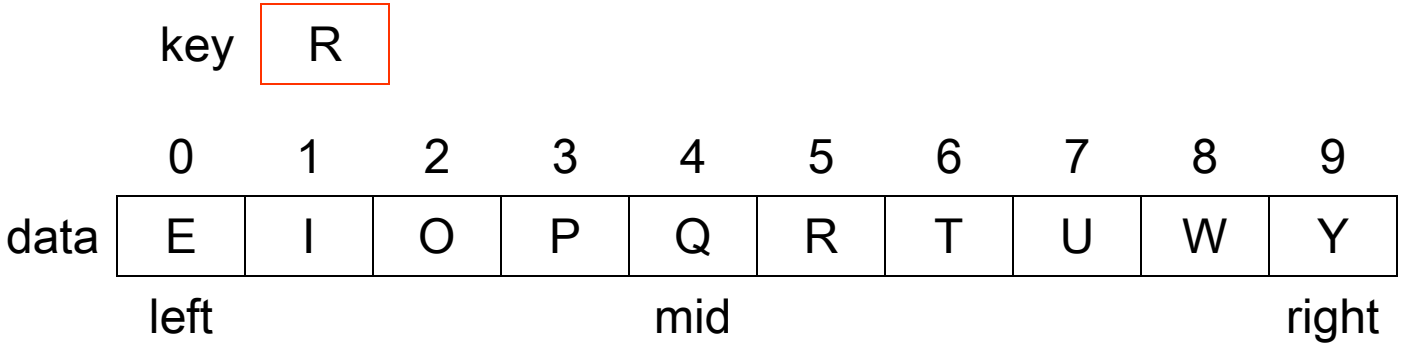
Searching an Ordered Array (cont.)

```
// binarySearch():  searches a sorted list for a key
public static int binarySearch(char[] data, char key){
   int left = 0;
   int right = data.length-1;
   while (left <= right) {
        int mid = (left + right)/2;
        if (data[mid] == key) {
                return mid;
        }
        else if (data[mid] < key) {
                left = mid + 1;
        }
        else {
                right = mid – 1;
        }
   }
   return –1;  //not found
}
```


Searching an Ordered Array (cont.)

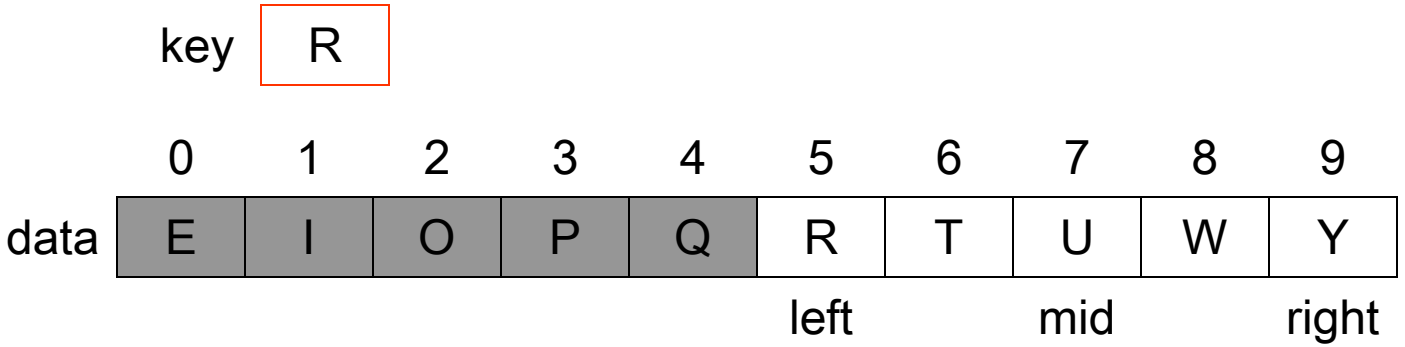


1: initial state



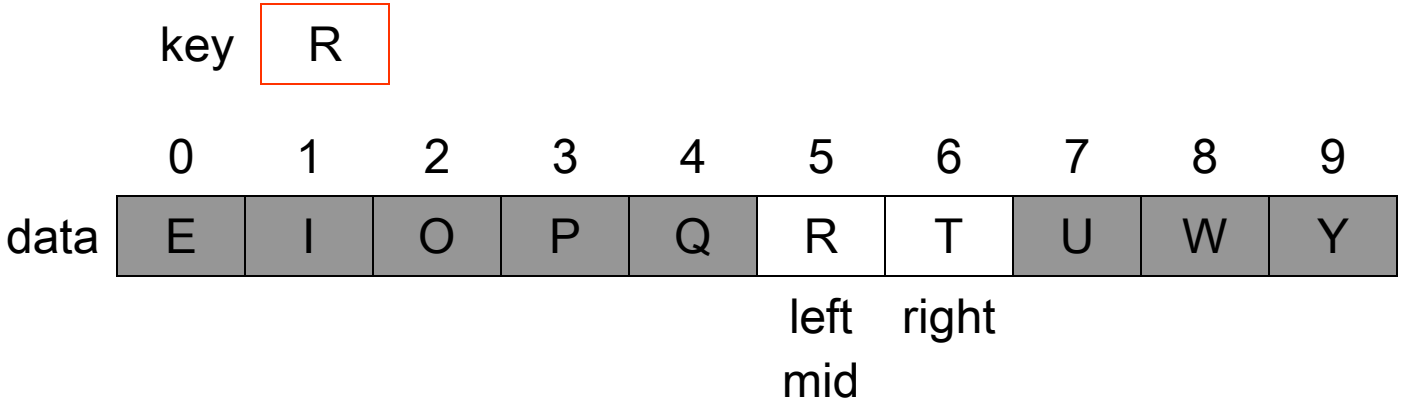
2: enter while loop and assign mid

Searching an Ordered Array (cont.)



3: after comparison and reset – shaded elements no longer considered

---



4: data[mid] == key (stop with success)

More on Passing Arrays as Parameters

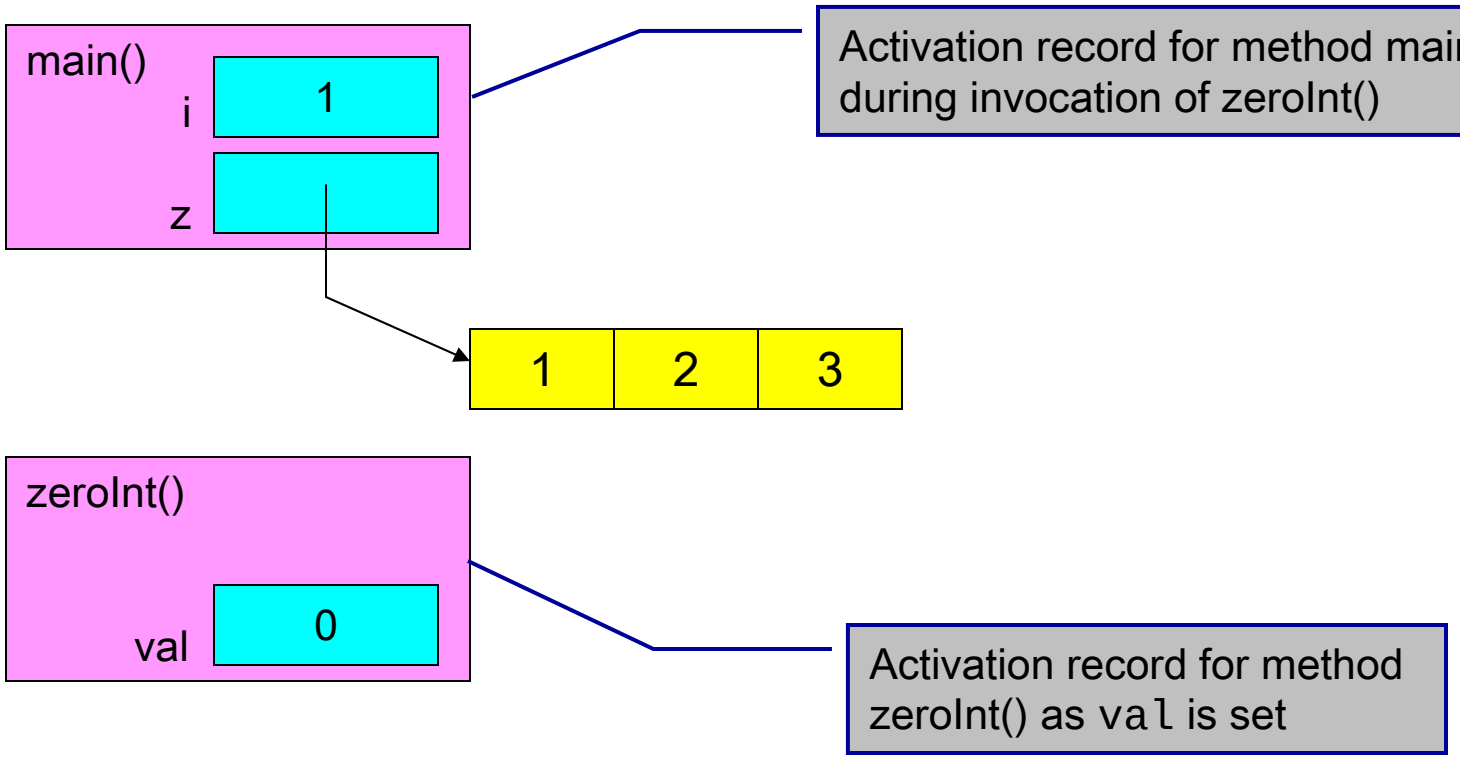
```
// demonstrates array parameter nuances
public class ArrayDemo{
   public static void main(String[] args {
        int i = 1;
        int[] x = {1, 2, 3};
        System.out.println("int i: " + i);
        System.out.println("array z: " + z[0] + " " + z[1]
              + " " + z[2]);
        zeroInt(i);
        zeroArray(z);
        System.out.println("int i: " + i);
        System.out.println("array z: " + z[0] + " " + z[1]
              + " " + z[2]);
   }
```


More on Passing Arrays as Parameters (cont.)

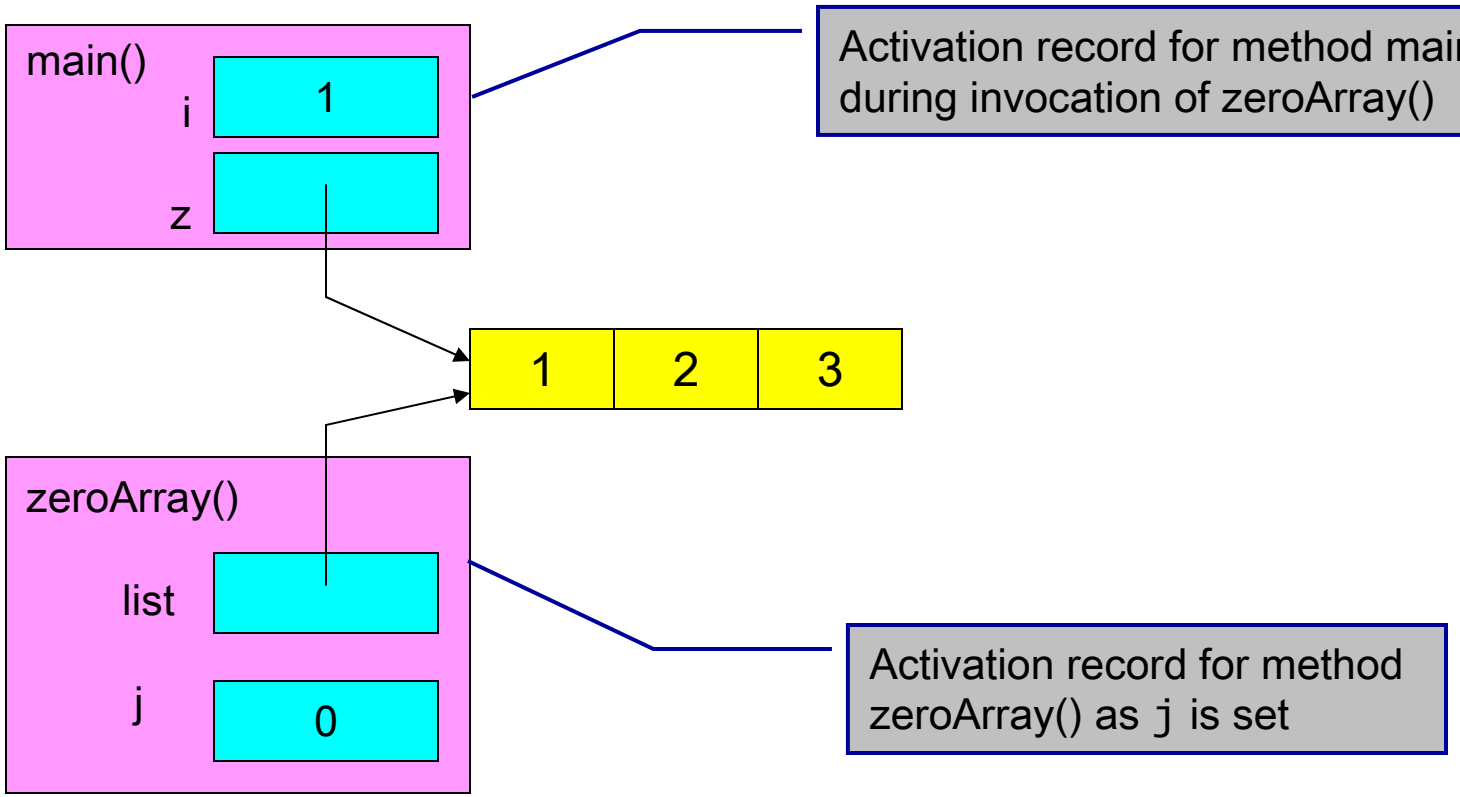
```
    public static void zeroInt(int val) {
        val = 0;
    }

    public static void zeroArray(int[] list) {
        for (int j = 0; j < list.length; ++j){
                list[j] = 0;
        }
    }
}//end class ArrayDemo
```

Output: int i: 1
array z: 1 2 3
int i: 1
array z: 0 0 0 **WHY?**

More on Passing Arrays as Parameters (cont.)



More on Passing Arrays as Parameters (cont.)



More on Passing Arrays as Parameters (cont.)