Java Arrays

Thumrongsak Kosiyatrakul
tkosiyat@cs.pitt.edu

Java Arrays

- So far, for the most part, we have stored data in a one-to-one fashion
  - One variable, one value (or object)
- This works fine if we know **exactly how many values** we will need to store, and if **threr are few of them**
- However, consider the following scenario:
  - We want to input the test scores of a given number of students and then:
    1. find the maximum,
    2. find the minimum,
    3. calculate the average, and
    4. list them in sorted order

- We can do the first three using only a few variables

```
// Not a compilable code...Just idea
// first score
score = inScan.nextInt();
sum = score;
min = sum;
max = sum;
// other scores
while(...) {
    score = inScan.nextInt();
    sum += score;
    if(score > max)
        max = score;
    if(score < min)
        min = score
}
average = (double) sum / count;
```

- However, what about listing them in **sorted order**?

Java Arrays

- We cannot know the final order **until** all scores have been read
  - Last value could be the smallest, the largest, or anywhere in between
- Thus, we need to **store all of the values** as they are being read in, **then sort them** and print them out
- TO do this, we need a good way to store an **arbitrary number of values**, without requiring the same number of variables
  - This is a good example of where an **array** is necessary

Java Arrays

- In Java, arrays are **objects**, with certain properties
  - Like other reference types
- Simply put, **an array is logically a single variable name that allows access to multiple variable locations**
- In Java, the locations also must be
  - **contiguous**: Each directly follows the previous in memory, and
  - **homogeneous**: all references (or value) in the array are of the same type

Syntax of Java Arrays

- First, consider only **primitive type** data
  - We create a Java array in two steps:
    1. Declare a variable of type array of a primitive type:

```
prim_type[] var_name;
```

- `prim_type` is any primitive type
- `var_name` is any legal identifier
- This creates an array variable but **not** an actual array object

    2. Construct an array object of the primitive type:

```
var_name = new prim_type[arr_size];
```

- `arr_size` is the number of elements that will be in the array (must be greater than or equal to zero)
- `arr_size` can also be a variable or an expression
- Indexing in Java always starts at 0
- This **creates the array object**

Arrays Examples

- Examples:

```
int a = 20, b = 30;
int[] x, y, z;
x = new int[10];
y = new int[a];
z = new int[a + b + 12];
```

- Can be done in one line:

```
double[] values = new double[50];
```

- Once we have created the array, we now can use it but keep in mine that:
  - numeric types are initialize to 0
  - boolean type are initialize to `false`

Indexing an Array

- An array variable gives us access to the **beginning** (first element) of the array
- To access an individual location (element) in the array, we need to index, using the [] operator
- Example:

```
int[] myArray = new int[20];
myArray[5] = 123;
myArray[10] = 2 * myArray[5];
myArray[11] = myArray[10] - 1;
myArray[20] = 5;        // ArrayIndexOutOfBoundsException
```

- Be careful, you can only use what you have created
  - From the above example, array size is 20 elements
  - The first element is at index 0
  - The last element is at index 19 (not 20)
- Index of an array must be greater than or equal to 0

Iterating through an Array

- We can easily iterate through an entire array using a loop (often a `for` loop)
- To know "when to stop" we access the length attribute of the array variable

```
int[] myArray = new int[25];
int myArrayLength = myArray.length; // myArrayLength will be 25
```

- Example:

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

  - Note that there is no `()` after `myArray.length`
  - `.length` only tells you the number of elements that the array has
    - It does not tell you how many elements in the array that contains data

Iterating through an Array

- Let's look at this code again:

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

- Another way to iterate through the array is as follows:

```
for(int aValue : myArray)
{
    System.out.println("Value: " + aValue);
}
```

  - Note that there is no counter in the above example
  - aValue will be changed to the next element in the array at every iteration

Direct Access and Sequential Access

- The previous two slides demonstrate the two basic ways of accessing arrays
  - **Direct Access**
    - Arbitrary item are accessed by providing the appropriate index of the item
  - **Sequential Access**
    - Items are accessed in index order from beginning to end (or from end to beginning)
  - The usefulness of arrays comes from allowing access in both of these way
  - Let's see both direct and sequential access of arrays with a file example
    - See `ex11.java`

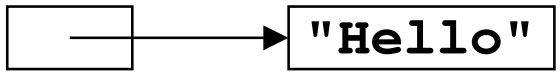
References and Reference Types

- Recall from previous discussion that Java has **primitive types** and **reference types**
  - **Primitive types**: Data values are stored directly in the memory location associated with a variable

```
int x = 12;
```

  - **Reference types**: Values are references to objects that are stored elsewhere in the memory

```
String s = "Hello";
```

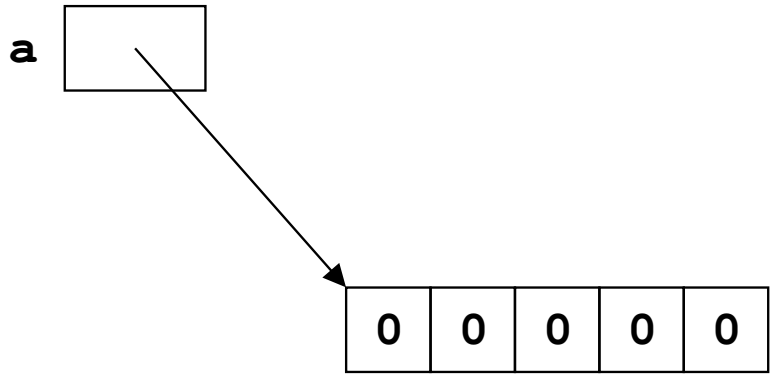


Array as Reference Types

```
int[] a;
```

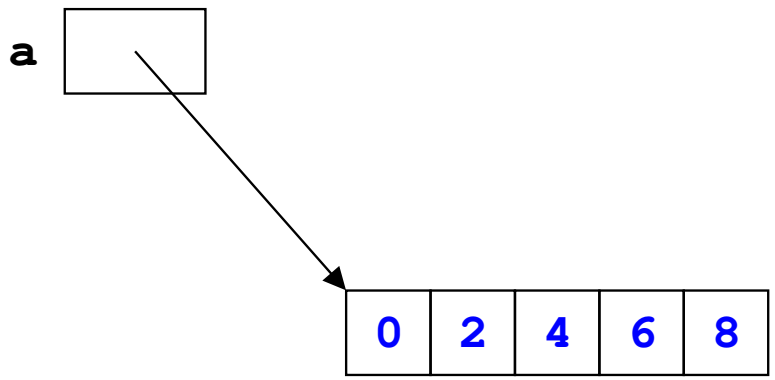
Array as Reference Types

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



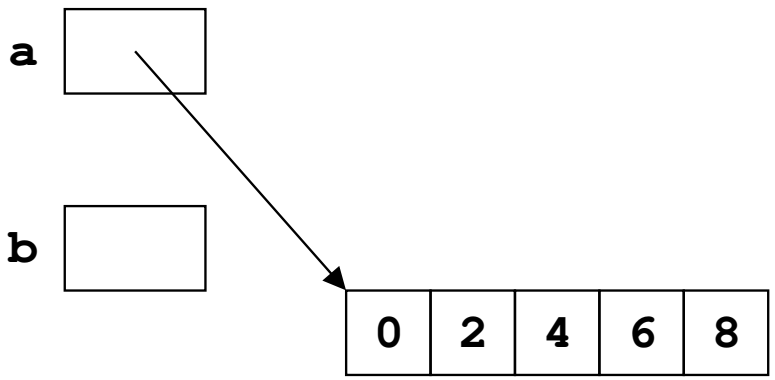
Array as Reference Types

```
int[] a;
a = new int[5];
for(int i = 0; i < 5; i++)
    a[i] = i * 2;
```



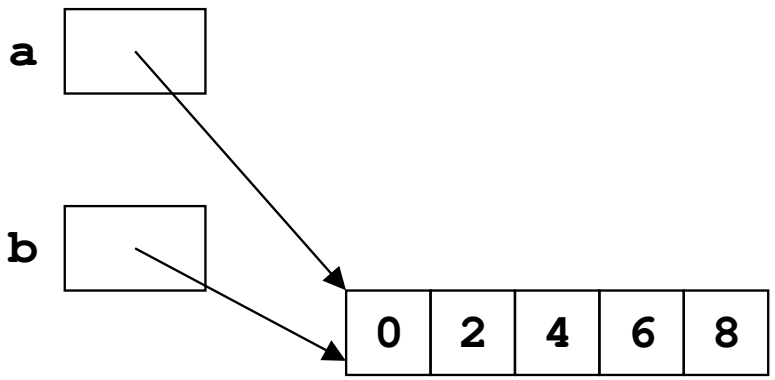
Array as Reference Types

```
int[] a;
a = new int[5];
for(int i = 0; i < 5; i++)
    a[i] = i * 2;
int[] b;
```



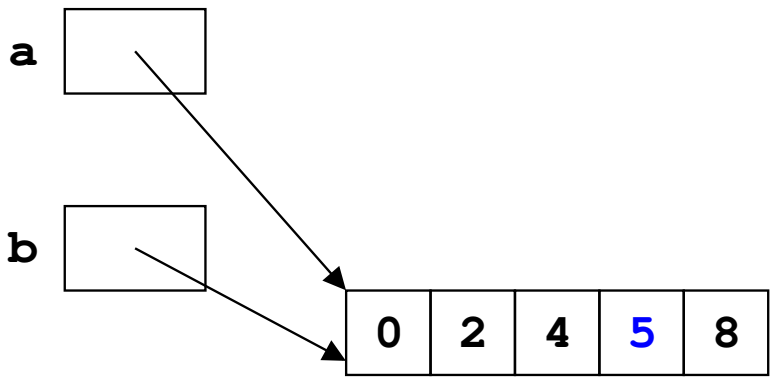
Array as Reference Types

```
int[] a;
a = new int[5];
for(int i = 0; i < 5; i++)
    a[i] = i * 2;
int[] b;
b = a;
```



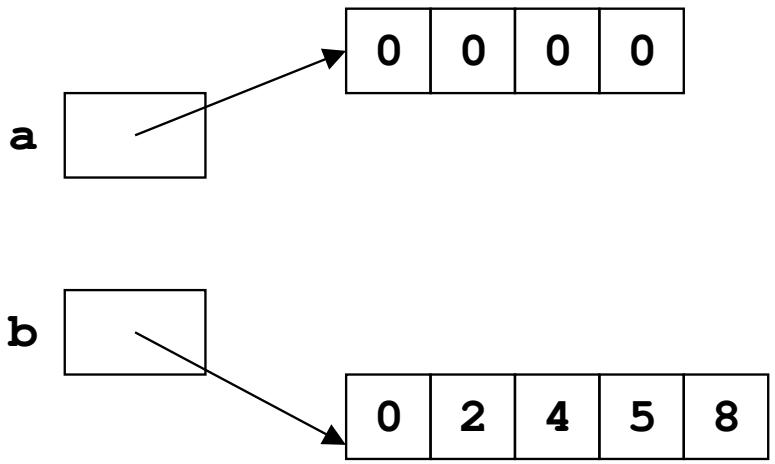
Array as Reference Types

```
int[] a;
a = new int[5];
for(int i = 0; i < 5; i++)
    a[i] = i * 2;
int[] b;
b = a;
a[3] = 5;
```



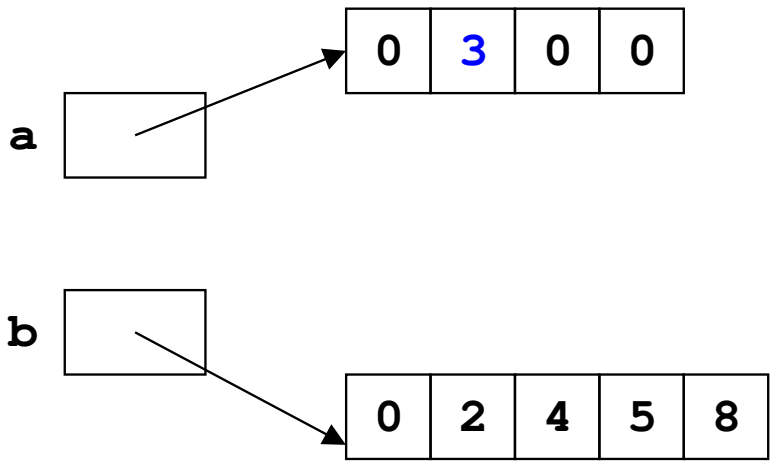
Array as Reference Types

```
int[] a;
a = new int[5];
for(int i = 0; i < 5; i++)
    a[i] = i * 2;
int[] b;
b = a;
a[3] = 5;
a = new int[4];
```



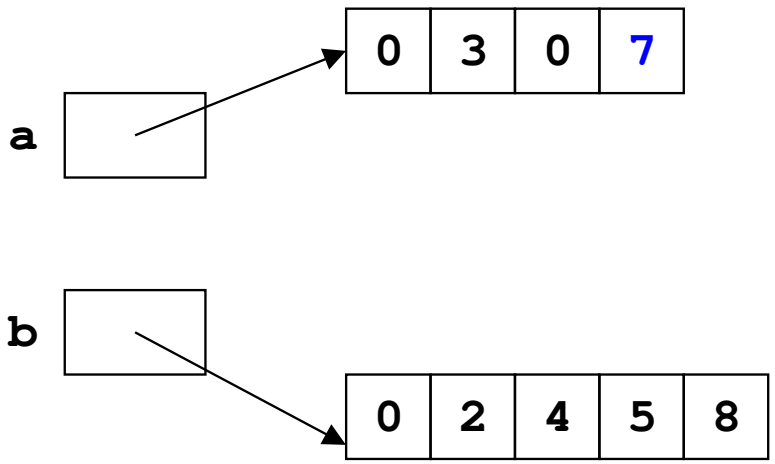
Array as Reference Types

```
int[] a;
a = new int[5];
for(int i = 0; i < 5; i++)
    a[i] = i * 2;
int[] b;
b = a;
a[3] = 5;
a = new int[4];
a[1] = 3;
```



Array as Reference Types

```
int[] a;
a = new int[5];
for(int i = 0; i < 5; i++)
    a[i] = i * 2;
int[] b;
b = a;
a[3] = 5;
a = new int[4];
a[1] = 3;
a[3] = 7;
```



Arrays as Parameters

- As we discussed earlier, all Java parameters are **values**
  - A copy of the argument is passed to the parameter
  - Changes to the parameter do not effect the argument
- What about arrays?
  - Since an array in Java is an object by executing

```
int[] myArray = new int[10];
```

    - the value stored in the variable `myArray` is a number
    - the number is a location (address) of the object
  - Using an array as an argument of a method, it is still passed by value
    - The value (number) that is copied to the method is the **reference**/location of the object but not the object itself

Arrays as Parameters

- Since the parameter of the method becomes the **reference**/location of the array, the method can
  - read data from the array
  - write data to the array
  - modify data in the array
  - sort the array
- Example:

```
public static void main(String[] args) {
    int[] nums = {1, 2, 3, 4, 5};
    myMethod(nums);
    :
}
public static void myMethod(int[] anArray) {
    anArray[0] = 9;        // nums[0] in main() is also 9
    anArray = new int[2]; // nums still refers to the old array
}
```

  - A methods cannot change the argument variable in the method
- See `ex11.java`

Searching an Array

- Often we may want to see if a value is stored in an array or not
  - Is this book in the library?
  - Is John Smith registered for a class?
- There are many searching algorithms available. Some simple, some sophisticated.
- We will start off with a simple one called **Sequential Search**

Sequential Search

- **Sequential Search** starts at the beginning of the array and check each item in sequence until the end of the array is reached or the item is found
  - Note that we have two important conditions here:
    - One stops the search with failure (get the end of the array)
    - The other stops the search with success (found the item)
  - We should always consider all possible outcomes when developing algorithms
- Question: What kind of loop is best for this?
  - Think about what needs to be done
- Look at example `ex12a.java`

Arrays of Objects

- We have created and use Java arrays of primitive types:

```
int[] data;             // declare variable (reference)
data = new int[20];     // create array object
:
data[4] = 77;           // index array to access locations
```

- How does it differ if we want **arrays of objects**
  - The first two steps are the same:
    - Declare variable
    - Create array object
  - Example:

```
String[] names;          // declare variable (reference)
names = new String[15];  // create array object
```

Arrays of Objects

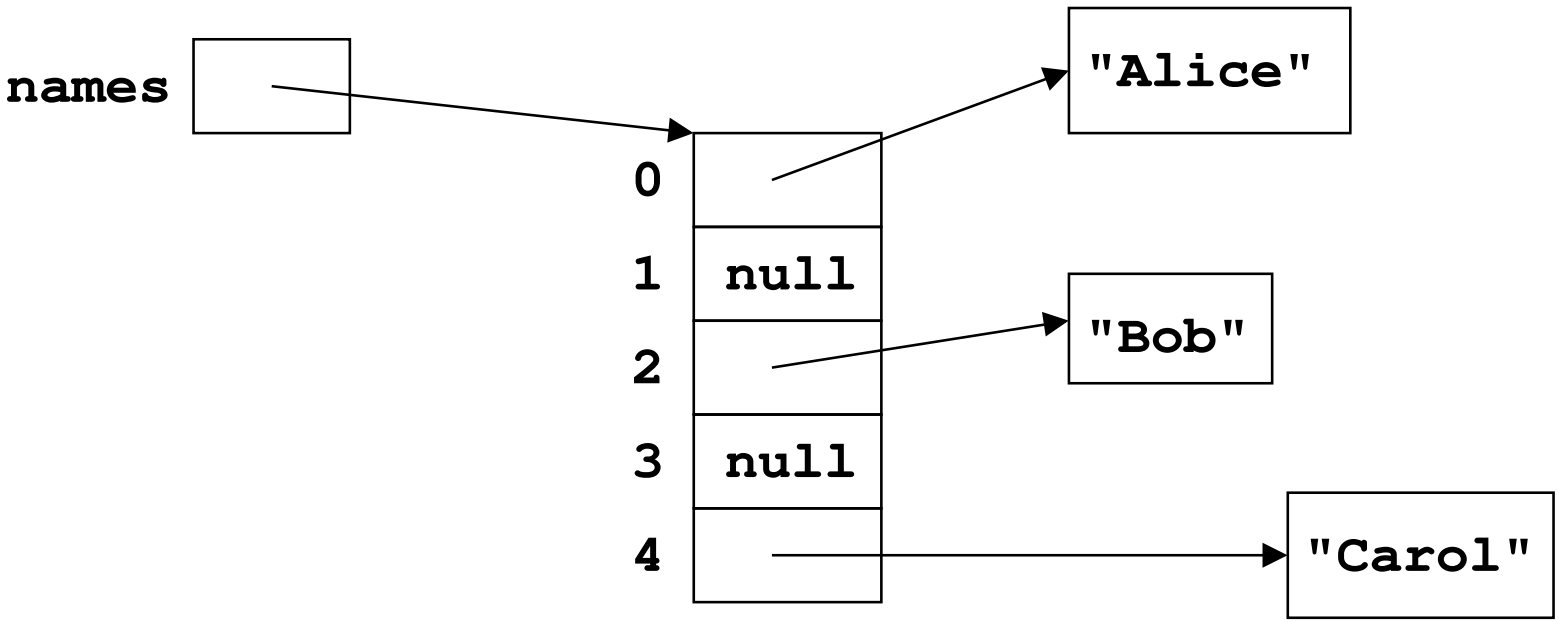
- However, remember that objects are accessed by **reference types**
- Thus, when we create the array, we have an **array of references**, with no object yet
  - All of the locations are **initialized to** `null`
    - `names[0]` is `null`, `names[1]` is `null`, and so on
  - We need to create objects to store in the array separately
- For example:

```
String[] names = new String[5];
names[0] = new String("Alice");
names[2] = new String("Bob");
names[4] = new String("Carol");
```

  - Note that `names[1]` and `names[3]` are still `null`

Arrays of Objects

- Note that we have two levels of references here:



Instance Data and Composition

- When we create a new class, we can have arbitrary instance variables within it.
  - If the instance variables are reference types (i.e. other classes), we say we are building a new class via **composition**
    - We are "composing" the new class from pieces that already exist, putting them together in an appropriate way
    - We briefly discussed this already with the `PlayList` class
    - Also sometimes called **aggregation**
    - Our use of these classes is limited to the functionality provided as public
    - We are building new classes using "off the shelf" components, so we may have to compromise based on what the "off the shelf" components can do

Arrays as Instance Data

- For example, if an **array** is used as an instance variable
  - We have the same access to the array **within our class** as we would anywhere else in our program
  - However, from **outside the class**, we may not even know the array is being used
    - **Encapsulation and data hiding**
  - See `ex12b.java` and `Scores.java`
- Yet another example of composition is seen in our previous example, `PlayList.java`
  - From outside the `PlayList` class, we do not even necessary know that class `Song` is being used within `PlayList`

Resizing an Array

- Java array objects can be of any size
  - However, once created, **they cannot be resized**
  - This is fine if we know how many items we will need in advance:

```
System.out.println("How many integers? ");
int size = inScan.nextInt();
int[] data = new int[size];
```

  - However, we do not always know this in advance
    - User may have an arbitrary amount of data and does not know how much until he/she has entered it
    - Mount may vary over time (e.g., the number of students in a University)

Resizing an Array

- So, what do we do if we fill our array?
  - **Logically**, we must "resize" it
  - **Physically**, we must do the following:
    1. Create a new, larger array object
    2. Copy the data from the old array to the new
    3. Assign our reference to the new object
  - This is not difficult syntactically, but it is important to realize that **this takes time, especially if the array is large**
  - Clearly, we do not want to do this too often
  - A typical approach is to **double the size**, so we have a lot of free location after the resizing
    - Why **double** the size?, take CS0445

Resizing an Array

- What if we do not have enough data to fill all of those new slot?
  - We must **keep track of the number of locations that are actually being used in the array**
    - We need an additional variable besides the array data itself
  - This way, we can "add" elements to the end of the array until it fills – only then will we have to resize
  - Note that the **array size** and **number of elements** being stored in the array are **not necessarily the same**
  - This is what is done in the predefined `ArrayList` class
    - We will learn about it soon

ArrayList

- Programmers can use arrays in arbitrary ways
  - However, many applications require a common set of array operations
    - Add an object to the end of an array
    - Find an object in an array
    - Iterate through an array
    - Remove items from an array
  - Rather than making the programmer implement these operations each time they are needed, the developers of Java have include a standard class that already does them
- The `ArrayList` class

ArrayList

- Remember **data abstraction**?
  - We can use an `ArrayList` object effectively without having to know how it is implemented
    - We do not need to know the internal data representation
    - We do not need to know the method implementation
      - When and how it is resized?
  - We simply need to look up its functionality in the Java API
- However, it is useful for computer scientists to understand how the `ArrayList` is implemented
  - Helps us to better understand programming in general
  - Helps us to implement similar types if necessary
- Look at a simple example: `ArrayL.java`

ArrayList

- Idea:
  - Data is maintained in two parts:
    - an **array** to actually store the information
    - an **int** to keep track of the number of elements being stored
  - Most of our operations are concerned with the **logical size** of the array
    - Number of actual elements being stored
  - The **physical size** of the array is **abstracted out of our view**
    - This changes as necessary but we never need to know what is actually is, in order to use the `ArrayList`
    - Recall the **resizing** discussed earlier

ArrayList

- We can also implement this type of variable size array ourselves if we want to
  - We may want to do this if our needed functionality is very different from that of the `ArrayList`
  - We simply need to keep an array and an `int` to keep track of the number of used locations
  - You will do a simple example of this is a Lab

Two-Dimensional Arrays

- A **two-dimensional array** is actually an array of arrays
- For example,

```
int[][] x = new int[4][8];
```

- First index (left) gives us an array of integers
  - `x[0]` is an array of integers with 8 elements
  - Sometimes we say `x[0]` is the first row, `x[1]` is the second row and so on
  - We say this is "row major order"
- The second index (right) gives us "column"
- To iterate through all locations, we typically use nested loop

```
for(int row = 0; row = x.length; row++)
{
    for(int column = 0; column = x[row].length; column++)
    {
        System.out.print(x[row][column] + " ");
    }
    System.out.println();
}
```

- See `ex13.java`