# ARRAYS

## OVERVIEW

# OVERVIEW

- **In many programs, we need to store and process a lot of data with the same data type**
  - Processing test scores to find the class average
  - Tabulating bank deposits and withdrawals
  - Displaying images on a computer screen

- **Arrays in Java give us a way to accomplish this goal**
  - Declare an array of desired data type and size
  - Store data values in each array location
  - Process data values to solve a specific problem

# OVERVIEW

- **How do we process data in arrays?**
  - Depends on needs of the application
  - Some applications create summaries of data in array
  - Some applications print a subset of data in array
  - Some applications search for data in the array
  - Some applications move data around in the array

- **How to we implement this array processing?**
  - Use iteration to loop over array elements
  - Use functions to simplify code reuse

# OVERVIEW

- **We need to learn a variety of array processing algorithms in order to become strong Java programmers**

- **Each array processing algorithm has its pros/cons**
  - Some have faster run times, some are slower
  - Some take more memory, some take less memory
  - Some are complex to implement, some are simple

- **To understand these differences, we must learn scientific methods for algorithm analysis and program testing**

# OVERVIEW

- **Lesson objectives:**
  - Learn the syntax for declaring arrays in Java
  - Learn how to store and process data in arrays
  - Learn how to search and sort data in arrays
  - Study example programs showing their use
  - Complete programming project using arrays

# ARRAYS

## PART 1

## ARRAY BASICS

# DECLARING ARRAYS

- **Arrays were invented to conveniently store multiple values of the same data type in one variable**
  - Picture an array as a long box divided into N slots


- **Array elements are stored in separate memory locations and accessed based on their position**
  - The first array element is in location 0
  - The last array element is in location N-1

# DECLARING ARRAYS

- **The syntax for an array declaration is:**

```
data_type array_name[ ] = new data_type [ array_size ];
```

**or**

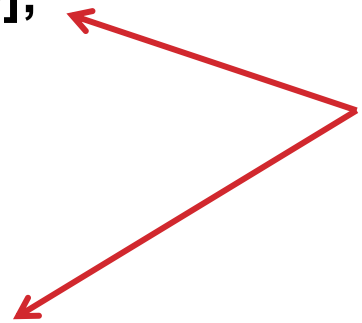
```
data_type[ ] array_name = new data_type [ array_size ];
```

**where**

data_type can be any Java type (int, float, etc.)
array_name follows Java variable name rules
array_size is an integer

# DECLARING ARRAYS

```
// Valid array declarations
float[ ] Data = new float[100];
int List[ ] = new int[20];

int Size = 50;
char Name[ ] = new char[Size];
```



The array size can be an integer constant or integer variable

# DECLARING ARRAYS

```
// Dynamic array declaration
int Size = 0;
while (Size <= 0)
{
   System.out.print("Enter array size: ”);
   Size = scanner.nextInt();
}
int Array[ ] = new int[Size];
```

# DECLARING ARRAYS

```
// Invalid array declarations
int Length = -1;
float Data[ ] = new float[Length];
int List[ ] = new int[31.75];
```

Array size must be a positive integer

You can not use floats to specify the array size

# DECLARING ARRAYS

- **The size of an array in memory is given by (number of elements in array) * (number of bytes for each element)**

**float Data[ ] = new float[100];**

- One float takes 4 bytes
- Array size is 100 * 4 = 400 bytes

**char Name[ ] = new char[20];**

- One char takes 1 byte
- Array size is 20 * 1 = 20 bytes

# ARRAY ACCESS

- **To access an array element, we need to give name of variable and the index (location) of desired element**
  - Eg: array_name[array_index]

- **In Java arrays are always "zero indexed"**
  - The first array element is at location 0
  - The last array element is at location N-1
  - If you attempt to use an array index that is outside the range 0..N-1 you will get a run time error

# ARRAY ACCESS

```
// Valid array access
float Data[ ] = new float[100];
…

Data[0] = 7;
Total = Total + Data[2];
System.out.println(Data[7]);
```

We use elements of an array just like any other variable, as long as the array index is within the range 0..SIZE-1

# ARRAY ACCESS

```
// Invalid array access
Data[4.3] = 28;
…
Data[-8] = 0.0;
Data[200] ++;
…
Data = scanner.nextFloat();
System.out.println(Data);
```

The array index can **not** be a float

An **array bounds error** will occur if the index is outside 0..99 range

We can **not** read or write a whole array at one time

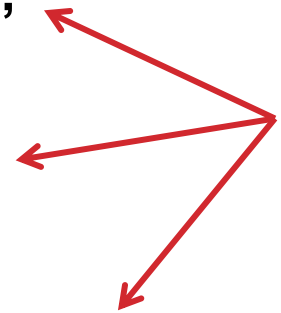
# ARRAY INITIALIZATION

- **Java arrays are automatically initialized**
  - Default value for integers and floats is 0
  - Default value for character is null
  - Default value for boolean is false
  - This is NOT true for many other languages (e.g., C++)

- **We can store initial values in an array at declaration time**
  - Give collection of N values to initialize array of size N

# ARRAY INITIALIZATION

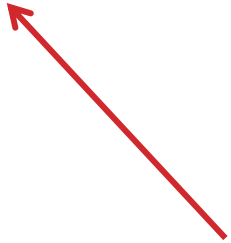
```
// Valid array initialization
int Value[ ] = {3,1,4,1,5,9,2,6,5,3};
…
char Name[ ] = {'J', 'O', 'H', 'N'};
...
float Scores[ ] = {93.5, 92.0, 90.1, 85.7, 83.3, 76.5};
```



The size of these arrays is determined by the number of values

# ARRAY INITIALIZATION

```
// Invalid array initialization
int Data[ ] = new int[10];
Data = {2,1,3,1,4,1,5,1,6,1};
```



We are **not** allowed to initialize an array with a sequence of values after it has been created

# ARRAY INITIALIZATION

- **We can initialize an array using the "fill" function in the Arrays class as follows**

```
// Declare an array
int Data[ ] = new int[10];
…
Array.fill(Data, 42);
...
Arrays.fill(Data, 3, 6, 17);
```

The initial values in array are all 0

This sets all of the array values to 42

This sets locations 3..5 to value of 17

# ARRAY LENGTH

- **It is important to know how many data values are in an array in order to process this data**
  - One approach is to keep track of the array size in an integer variable when we create the array

```
int Size = 50;
float Data[ ] = new float[Size];
```

- **What happens if you change the value of Size?**
  - You could easily have an array bounds error when processing the array

# ARRAY LENGTH

- **The number of data values in every Java array is stored in the "length" attribute of the array when it is created**
  - To get this value we add **.length** after the array name

```
float Data[ ] = new float[100];
int index = scanner.nextInt();
if ((index >= 0 && (index < Data.length))
   Data[index] = 42;
```

The value of Data.length will be 100 in this case

# COPYING AN ARRAY

- **Unfortunately we can NOT copy an array using just one assignment statement**

```
// Declare an array
float Data[ ] = new float[10];
float Copy[ ] = Data;
…
Copy[3] = 42;
…
System.out.println(Data[3]);
```

The Copy variable refers to the same array as Data

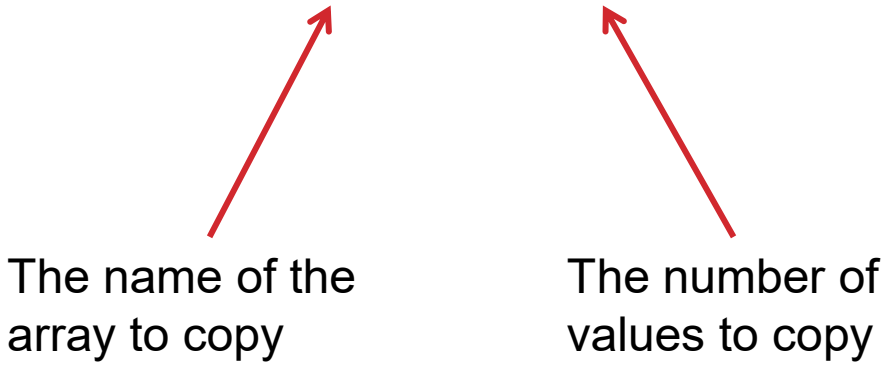
This will store 42 in Data[3]

This will print 42

# COPYING AN ARRAY

- **We can copy data from one array into another array using the "copyOf" function in the Arrays class**

```
// Declare an array
float Data[ ] = new float[10];
float Copy[ ] = Arrays.copyOf(Data, data.length);
```



# ARRAYS AND LOOPS

- **It is very natural to use loops to process arrays**
  - Read N input values into an array
  - Write N output values from an array
  - Calculate total of N values in array

- **We must take care to stay within array bounds**
  - Never use index less than 0
  - Never use index greater than N-1
  - If you do go outside this 0..N-1 range, it may cause a "array access error" error at run time

# ARRAYS AND LOOPS

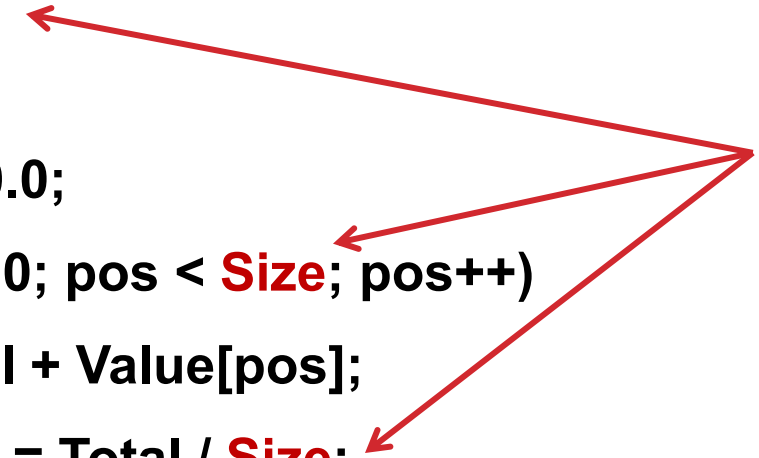
```
// Input output example
float Data[ ] = new float[10];

for (int i = 0; i < 10; i++)
   Data[i] = scanner.nextFloat();

for (int i = 0; i < 10; i++)
   System.out.println(Data[9-i]);
```

Loop to read 10 values into the Data array

Loop to write 10 Data values in reverse order

# ARRAYS AND LOOPS

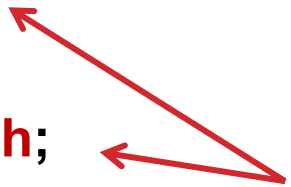
```
// Average calculation example
int Value[ ] = {3,1,4,1,5,9,2,6,5,3};
int Size = 10;


float Total = 0.0;
for (int pos = 0; pos < Size; pos++)
   Total = Total + Value[pos];
float Average = Total / Size;
```



Here we store the length of the array in a variable and use this value the array processing loop

# ARRAYS AND LOOPS

```
// Average calculation example
int Value[ ] = {3, 1,4,1,5,9, 2,6,5,3,5, 8,9,7,9,3};

float Total = 0.0;
for (int pos = 0; pos < Value.length; pos++)
   Total = Total + Value[pos];
float Average = Total / Value.length;
```



It is easier and safer to use array length attribute in the array processing loop

# CONCISE ARRAY ACCESS

- **Java provides a very concise way to access array data inside a for loop using the following syntax:**

```
for ( data_type element : array )
{
    // block of statements to be repeated
}
```

- **This loop will iterate N times if there are N items in the array, and each time through the loop it will set "element" to be equal to the corresponding "array[index]"**

# CONCISE ARRAY ACCESS

```
// Average calculation example
int Value[ ] = {3,1,4,1,5,9,2,6,5,3};
int Total = 0;

for ( int Num : Value )
   Total += Num;

float Average = Total / Value.length;
System.out.println(Average);
```



The value of Num will go from Value[0] to Value[9] as the loop executes

# CONCISE ARRAY ACCESS

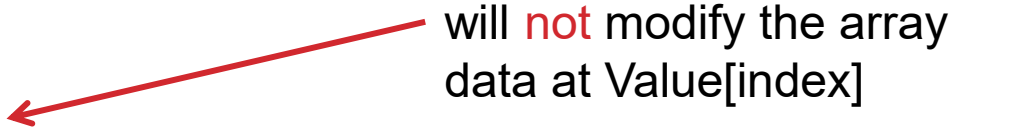
```
// Average calculation example
int Value[ ] = {3,1,4,1,5,9,2,6,5,3};
int Total = 0;

for ( int i = 0; I < Value.length; i++)
{  int Num = Value[i];
   Total += Num;
}
float Average = Total / Value.length;
System.out.println(Average);
```

This code is equivalent to the previous for loop

# CONCISE ARRAY ACCESS

- **We are allowed to access array contents using this for loop, but we can not modify the array contents using the "element" variable**

```
// Example of incorrect usage
int Value[ ] = {3,1,4,1,5,9,2,6,5,3};
for ( int Num : Value )
{
  Num = Num + 42;
  System.out.println(Num);
}
```



# CODE DEMO

**Array.java**

**MaxMin.java**

# USING PARTIAL ARRAYS

- **What happens if we do not know number of data values we need to store in an array in advance?**
  - Solution to declare a large array and use only part of it

- **How do we do this?**
  - We guess the maximum size needed for the array
  - Declare the array to be the maximum size needed
  - We use only part of this array to store our data
  - We also keep track of how much of the array is currently being used in a “Count” variable

# USING PARTIAL ARRAYS

- **Example: Reading student grades into an array**
  - Assume the user knows how many grades they will enter
  - Prompt the user for the grade count
  - Read the grade count from the user
  - Loop reading grades into array
  - Process the grade array in some way

- **Sample input:**

  5

  78 85 91 88 94

# USING PARTIAL ARRAYS

```
// User enters array count followed by grades
int SIZE = 1000;
float Grades[ ] = new float[SIZE];
```

We are guessing that the user will never want to use more than 1000 values

```
System.out.print("Enter count: ”);
int Count = scanner.nextInt();
```

We ask the user for how much of the array they want to use today

# USING PARTIAL ARRAYS

```
if (Count > SIZE)
   Count = SIZE;


for (int i = 0; i < Count; i++)
{
     System.out.print("Enter grade: ”);
     Grade[i] = scanner.nextFloat();
}
// Process the grade array here
```

We do error checking to make sure Count <= 1000

We can now loop from 0 up to Count-1 reading data from the user into the array

# USING PARTIAL ARRAYS

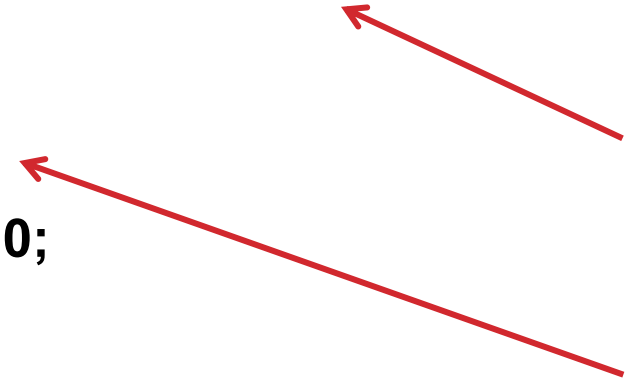
- **Example: Reading student grades into an array**
  - Assume the count is not known in advance and the grade data will be followed by a sentinal value of -1
  - Read first data value from user
  - While data value is not equal to the sentinal value
    - Store the grade in array
    - Read the next data value from user
  - Process the grade array in some way

- **Sample input:**

  78 85 91 88 94 -1

# USING PARTIAL ARRAYS

```
int SIZE = 1000;
float Grade[ ] = new float[SIZE];

int Count = 0;
float Input = 0.0;
```

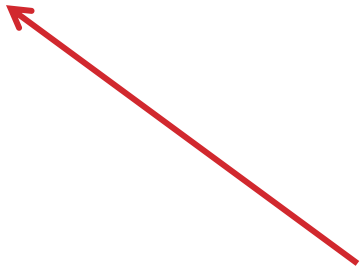


We declare an array that can hold up to 1000 grades

We use Count to keep track of how many were actually read

# USING PARTIAL ARRAYS

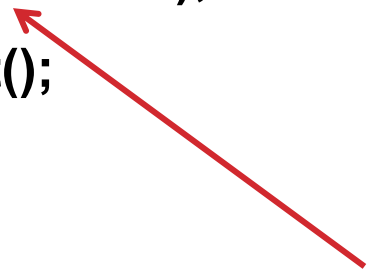
```
// User enters grades followed by -1 sentinel value
System.out.print("Enter grade: ”);
Input = scanner.nextFloat();
while ((Input != -1) && (Count < SIZE))
{
   Grade[Count] = Input;
   Count = Count + 1;
   System.out.print("Enter grade: ”);
   Input = scanner.nextFloat();
}
```



We stop reading user input when 1000 values are entered OR when the user types the -1 sentinel value

# USING PARTIAL ARRAYS

```
// User enters grades followed by -1 sentinel value
while ((Input != -1) && (Count < SIZE))
{
   System.out.print("Enter grade: ”);
   Input = scanner.nextFloat();
   Grade[Count] = Input;
   Count = Count + 1;
}
Count = Count - 1;
```



We stop reading user input when 1000 values are entered OR when the user types the -1 sentinel value

# SUMMARY

- In this section, we saw how to declare, initialize and access arrays in Java
- We also saw how loops could be used to read/write and process arrays in different ways
- Finally, we discussed how arrays can be used to store and process a variable number of elements

# ARRAYS

## PART 2

## ADVANCED ARRAYS

# ARRAYS AS PARAMETERS

- **How do we declare array parameters?**
  - Add the characters [ ] after the array name in the parameter declaration to tell compiler this is an array
- **How do we use array parameters?**
  - Just give the name of the array in the function call
- **What type of parameter is this?**
  - Arrays are treated as reference parameters to functions
  - Changes to an array in a function will change the array that was passed into the function (unlike value parameters)

# ARRAYS AS PARAMETERS

- **How can we add one to all values in an array?**

- **Write a function to process the array**
  - Declare array parameter
  - Loop over array in function doing operation

- **Call this function in the main program**
  - Pass in the name of the array
  - This array can me modified in function

# ARRAYS AS PARAMETERS

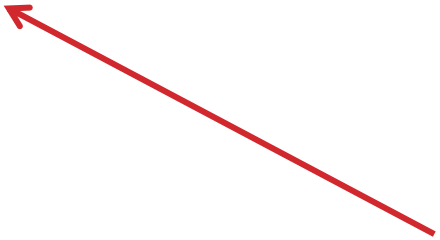
```
// Declare function to increment all values in an array
static void AddOne( float Value[ ] )
{
   for (int i= 0; i < Value.length; i++)
      Value[i] = Value[i] + 1.0;
}
```

We can call this function with float arrays of any size

We loop over all of the elements of the array and add 1.0 to each value

# ARRAYS AS PARAMETERS

```
// Call function to process array
float Data[ ] = {1,2,3,4,5,6,7,8,9,10};
AddOne( Data );
```



The function parameter is the name of the array we want to process “Data”

# ARRAYS AS PARAMETERS

- **How can copy data values from one array to another?**

- **Write a function to process the arrays**
  - Declare two array parameters
  - Loop over array in function doing operation

- **Call this function in the main program**
  - Pass in the names of both arrays
  - Only one array will be modified in this case

# ARRAYS AS PARAMETERS

```
// Declare function to copy array values
static void CopyData( float In[ ], float Out[ ] )
{
   for (int i=0; i < In.length; i++)
      Out[i] = In[i];
}
```

The array parameter "Out" will be modified

The array parameter "In" will not be modified

We loop over all of the elements of the "In" array and copy the value to "Out"

# ARRAYS AS PARAMETERS

```
// Declare function to copy array values
static void CopyData( float In[ ], float Out[ ] )
{
   for (int i=0; (i < In.length) &&
                 (i < Out.length); i++)
      Out[i] = In[i];
}
```

The array parameter "Out" will be modified

The array parameter "In" will not be modified

We loop over all of the elements of the "In" array and copy the value to "Out"

# ARRAYS AS PARAMETERS

```
// Call function to process array
float Data1[ ] = {1,2,3,4,5,6,7,8,9,10};
float Data2[ ] = new float[10];
CopyData( Data1, Data2 );
```

This function call will copy
Data1 data into Data2 array

# ARRAYS AS PARAMETERS

```
// Call function to process array
float Data1[ ] = {1,2,3,4,5,6,7,8,9,10};
float Data2[ ] = new float[ Data1.length ];
CopyData( Data1, Data2 );
```

This function call will copy
Data1 data into Data2 array

# ARRAYS AS PARAMETERS

```
// Call function to process array
float Data1[ ] = {1,2,3,4,5,6,7,8,9,10};
float Data2[ ] = new float[5];
CopyData( Data1, Data2 );
```

This function call will cause an array bounds error because output array is smaller than the input array

# ARRAYS AS PARAMETERS

```
// Call function to process array
float Data1[ ] = {1,2,3,4,5,6,7,8,9,10};
float Data2[ ] = new float[25];
CopyData( Data1, Data2 );
```

This function call will copy 10 values from Data1 into Data2 and remaining values in Data2 will be unchanged

# CHARACTER ARRAYS

- **Arrays of characters in Java can be used to store textual information (e.g. names, addresses, etc.)**

- **We declare and initialize character arrays as follows**

```
char [ ] name = {'J','o','h','n'};
```

- **We are not allowed to initialize with strings**

```
char [ ] name = "John";
```
X

- **We can print an entire character array in one command**

```
System.out.println(name);
```

# CHARACTER ARRAYS

- **Unfortunately, there are no methods to read individual characters or arrays of characters using the scanner**

```
char input = ' ';
char[ ] message = new char[100];


input = scanner.nextChar();
message = scanner.nextChar();
```

X

This code will not compile because there is no nextChar method

# CHARACTER ARRAYS

- **To read single characters in Java we must use the System.in.read() method**

```
char input = ' ';
input = (char) System.in.read();
```

  - This method actually returns the integer ASCII code for a character, so we have to cast it to a character using (char)
  - We also have to catch or throw IOExceptions when we call the read method

# CHARACTER ARRAYS

- **We can use a for loop to read an array of characters**

```
char[] message = new char[100];
for (int i=0; i<message.length; i++)
        message[i] = (char) System.in.read();
System.out.print(message);
```

  - This code will read 100 characters and save spaces and newlines in the message string (which may not be desired)

# CHARACTER ARRAYS

- **We can stop this for loop when end of line is reached**

```
char[] message = new char[100];
for (int i=0; i<message.length; i++)
{
        message[i] = (char) System.in.read();
        if (message[i] == '\n') break;
}
System.out.print(message);
```

Testing for the end of line character

# CHARACTER ARRAYS

- **Reading individual characters in Java is so bothersome that Java introduced the String data type**
  - Internally, the String data type has an array of characters to store the ASCII codes of all letters in a String
  - The String class also has many methods to manipulate the characters in this private array

- **It is very easy to read and write a Strings**

```
String name = scanner.next();
System.out.println(name);
```

# STRING LIBRARY

- **The developers of Java have created a very powerful library of string manipulation functions in the String class**

- **This class has functions in several categories**
  - Constructor functions
  - Access functions
  - Comparison functions
  - Search functions
  - Conversion functions

- **To use this library you must import java.lang.String**

# STRING LIBRARY

- **Constructor functions include:**
  - **String**(String s) – create a string with the same value as s
  - **String**(char[ ] a) – create a string that represents the same sequence of characters as array a

- **Access functions include:**
  - int **length**() – returns number of characters in the String
  - char **charAt**(int index) – returns the character at location index in the string
  - String **substring**(int beginIndex, int endIndex) – returns the string from index beginIndex through endIndex

# STRING LIBRARY

- **Comparison functions include:**
  - boolean **equals**(String str) – returns true if this string has the same sequence of characters as String str
  - int **compareTo**(String str) – compares two strings character by character lexicographically and returns
    - -1 if the first string is less than the second string
    - 0 if the two strings match each other
    - 1 if the first string is greater than the second string
  - int **compareToIgnoreCase**(String str) – compares two strings character by character lexicographically, ignoring case differences

# STRING LIBRARY

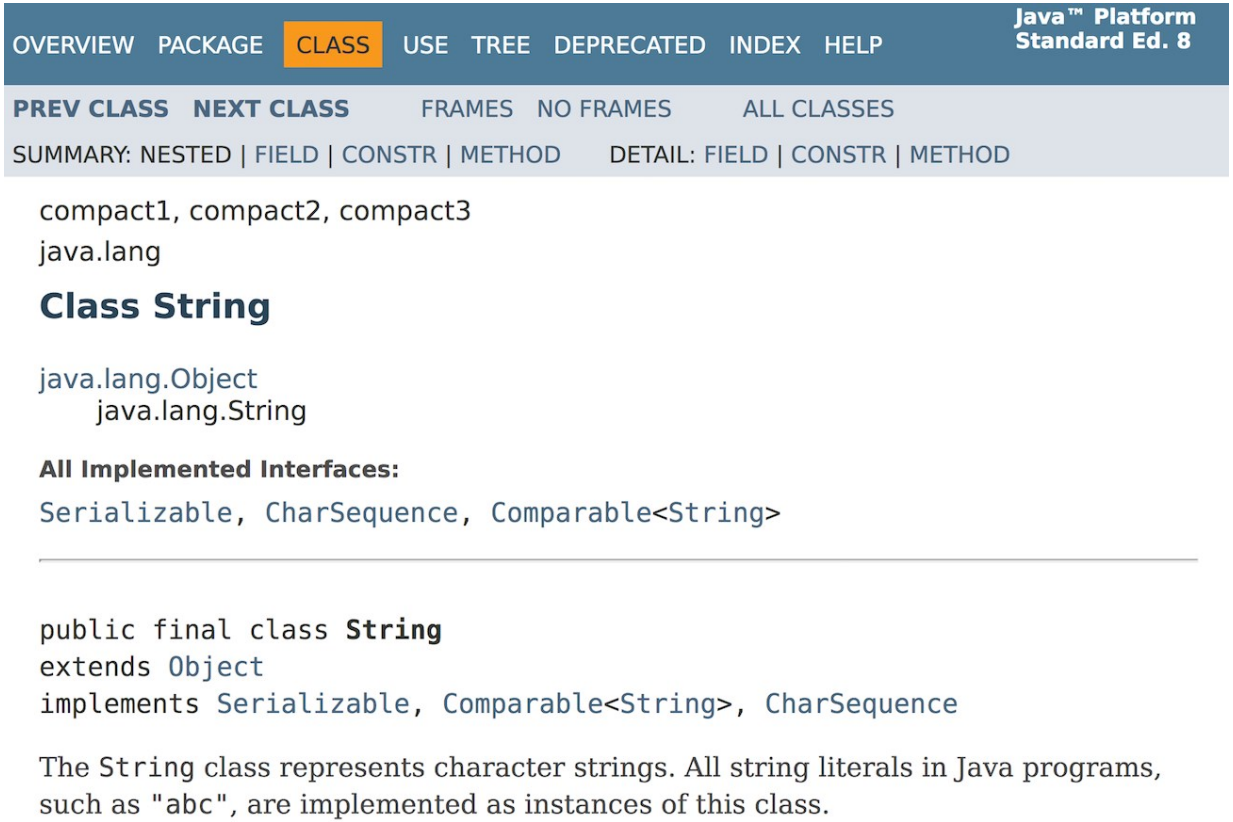
- **Search functions include:**
  - boolean **contains**(String substring) – returns true if this string contains the specified substring
  - boolean **startsWith**(String prefix) – returns true if this string starts with the specified prefix
  - boolean **endsWith**(String postfix) – returns true if this string ends with the specified postfix
  - int **indexOf**(String pattern) – returns the index of the first occurrence of the pattern string (or -1 if not found)
  - int **indexOf**(String pattern, int start) – returns the index of the first occurrence of the pattern string after the start index (or -1 if not found)

# STRING LIBRARY

- **Conversion functions include:**
  - String **toLowerCase**(String str) – returns this string with all characters converted to lower case
  - String **toUpperCase**(String str) – returns this string with all characters converted to upper case
  - String **replace**(String str1, String str1) – returns this string with each occurrence of str1 replaced with str2
  - String **trim**() – returns this string with all leading and trailing whitespace removed
  - String[ ] **split**(String delimiter) – returns an array of strings by extracting substrings from the input string that are separated by the delimiter (used to parse CSV files)

# STRING LIBRARY

- **See the official Java documentation here**



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.lang

## Class String

java.lang.Object
    java.lang.String

**All Implemented Interfaces:**

`Serializable, CharSequence, Comparable<String>`

---

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

The `String` class represents character strings. All string literals in Java programs, such as `"abc"`, are implemented as instances of this class.

# 2D ARRAYS

- **In many applications, data can be naturally organized as a two-dimensional grid of values**
  - Data in a spreadsheet
  - Pixels in an image

| | A | B | C |
|---|---|---|---|
| 1 | fred | bob | total |
| 2 | 3 | 1 | 4 |
| 3 | 4 | 1 | 5 |
| 4 | 5 | 9 | 14 |
| 5 | 2 | 6 | 8 |
| 6 | 5 | 3 | 8 |
| 7 | | | |

# 2D ARRAYS

- **Java will allow us to define 2D arrays by specifying**
  - The array name
  - The data type
  - The number of rows
  - The number of columns

```
// Example 2D array declaration
int Rows = 5;
int Cols = 3;
int A[ ][ ] = new int[ Rows ][ Cols ];
```

We must specify the number of rows and columns in this order

# 2D ARRAYS

- **We refer to 2D array locations using [row][column] index**
  - The rows are numbered 0..ROWS-1
  - The columns are numbered 0..COLS-1

← columns →

↓ rows →

| | | |
|---|---|---|
| A[0][0] | A[0][1] | A[0][2] |
| A[1][0] | A[1][1] | A[1][2] |
| A[2][0] | A[2][1] | A[2][2] |
| A[3][0] | A[3][1] | A[3][2] |
| A[4][0] | A[4][1] | A[4][2] |

# 2D ARRAYS

- **2D arrays can be initialized much like 1D arrays**
  - We must provide rows * columns values
  - We use curly brackets to group rows of values

**int Scores [ ][ ] = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };**

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

# 2D ARRAYS

- **Declaring 2D array parameters**
  - We use Data[ ][ ] when declaring a 2D array parameter
  - We can calculate the number of rows and columns in the 2D array using the .length attribute inside the function

```
int Rows = Data.length;
int Columns = Data[0].length;
```

- **Passing 2D arrays into functions as parameters**
  - When passing 2D array into a function we just use the name of the array (just like 1D arrays)

# 2D ARRAYS

- **Consider the problem of storing and displaying characters on an old fashioned VT52 computer terminal**
  - VT52s display 24 rows and 80 columns of characters
  - We need to store these characters in a 2D array

# 2D ARRAYS

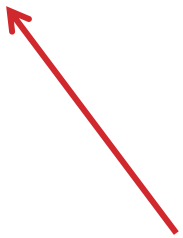
```
// Array declaration
int Rows = 24;
int Cols = 80;
char Screen[ ][ ] = new char[ Rows ][ Cols ];

// Array initialization
for (int r = 0; r < Rows ; r++)
   for (int c = 0; c < Cols ; c++)
      Screen[r][c] = ' ';
```

Here we declare a 2D array for the screen

Here we initialize the 2D array to all spaces

# 2D ARRAYS

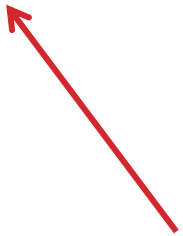
```
// Print array
for (int r = 0; r < Rows; r++)
{
   System.out.print("|");
   for (int c = 0; c < Cols; c++)
      System.out.print(Screen[r][c]);
   System.out.println("|");
 }
```



Here we print out characters one at a time in row column order

# 2D ARRAYS

```
// Print array
for (int r = 0; r < Rows; r++)
{
   System.out.print("|");
   System.out.print(Screen[r]);
   System.out.println("|");
 }
```



Since this is a character array we are allowed to print out a whole row at a time

# 2D ARRAYS

- **Consider the problem of processing student grades that are stored in a 2D array, with one row per student, and one column per homework assignment**
  - Student average = total of values in one row divided by number of homeworks (columns)
  - Homework average = total of values in one column divided by the number of students (rows)
  - Class average = total of values in the array and divided by the number of values in the array (rows * columns)

# 2D ARRAYS

```
// Array declaration
const int Students = 40;
const int Homeworks = 15;
float Grades[ ][ ] = new float[Students ][Homeworks ];

// Array initialization
for (int r = 0; r < Students ; r++)
   for (int c = 0; c < Homeworks ; c++)
      Grades[r][c] = scanner.nextFloat();
```

Here we declare a 2D array for the grades

Here we read the student scores to initialize the 2D arrray

# 2D ARRAYS

```
// Calculate homework average for one student
int student = 0;
float total = 0.0;
float average = 0.0;
System.out.print("Enter student index: ");
student = scanner.nextInt();
for (int c = 0; c < Homeworks ; c++)
    total = total + Grades[student][c];
average = total / Homeworks ;
System.out.println("Average = " + average);
```

We loop over one row in the 2D array to calculate the homework average for one student in class

# 2D ARRAYS

```
// Calculate class average for one homework
int homework = 0;
float total = 0.0;
float average = 0.0;
System.out.print("Enter homework index: ");
homework = scanner.nextInt();
for (int r = 0; r < Students ; r++)
    total = total + Grades[r][homework];
average = total / Students ;
System.out.println("Average = " + average);
```



We loop over one column in the 2D array to calculate class average for one homework assignment

# 2D ARRAYS

```
// Calculate class average on all homework
float total = 0.0;
float average = 0.0;
for (int r = 0; r < Students ; r++)
   for (int c = 0; c < Homeworks ; c++)
      total = total + Grades[r][c];
average = total / (Students *Homeworks );
System.out.println("Average = " + average);
```

We loop over all rows and columns in array to calculate class average

# SUMMARY

- In this section, we described how arrays of characters and Strings can be used to store and print text
- We also showed how 2D arrays can be defined and used to manipulate two-dimensional data

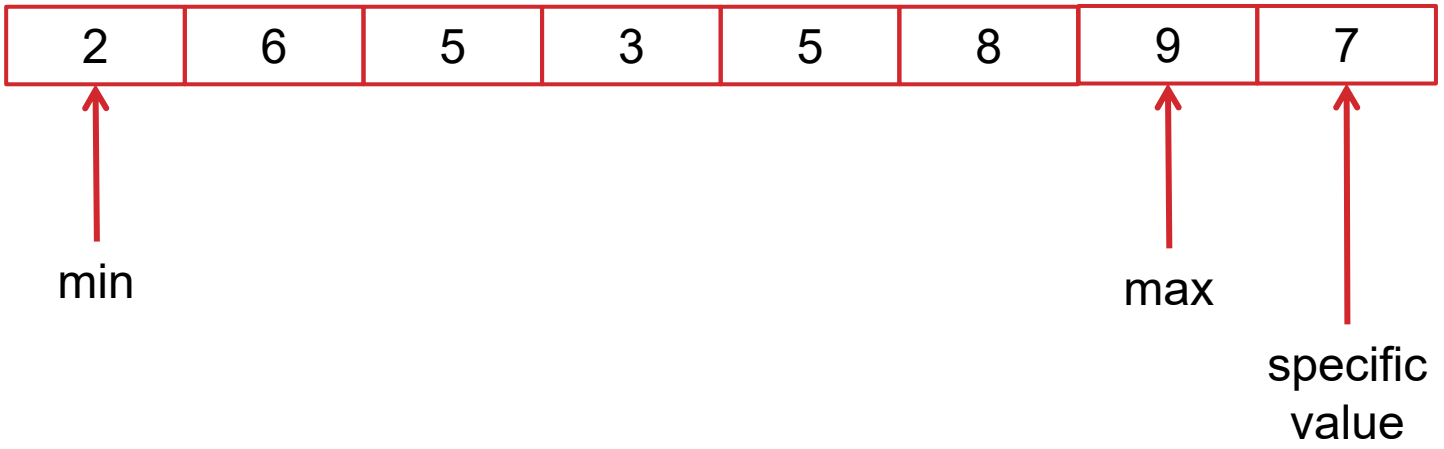
# ARRAYS

## PART 3

## SEARCHING AND SORTING

# LINEAR SEARCH

- **Once we have stored a collection of values in an array, we can search the array to answer a number of questions:**
  - Does a specific value (like 7) occur in array?
  - What is the maximum value in array?
  - What is the minimum value in array?



# LINEAR SEARCH

- **Linear search is the most basic algorithm for searching**
  - Start at beginning of array (index 0)
  - Look at each element of array one at a time
  - Check if we have found what we are looking for
  - Stop at end of the array (index N-1)
  - This process is typically implemented with a loop

# LINEAR SEARCH

```
// Linear searching for special value
float Special = 42;
for (int Pos = 0; Pos < Value.length; Pos++)
{
   if (Value[Pos] == Special)
      System.out.println("Found " + Special + " at " + Pos);
}
```

Loop over all array locations

Check for desired value in array

# LINEAR SEARCH

```
// Linear searching for max and min values
float Minimum = Value[0];
float Maximum = Value[0];
for (int Pos = 1; Pos < Value.length; Pos++)
{
   if (Value[Pos] < Minimum)
      Minimum = Value[Pos];
   if (Value[Pos] > Maximum)
      Maximum = Value[Pos];
}
```

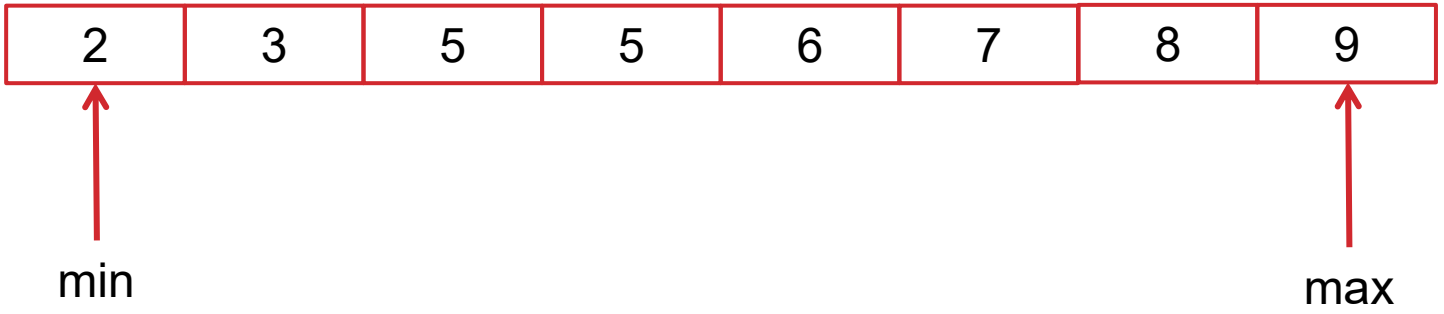
Initialize our best guess of min/max

Loop over all array locations

Update values of min/max as needed

# BINARY SEARCH

- **What happens if we are given an array with sorted values?**
  - Now we know exactly where min/max should be
  - Minimum value always at location 0
  - Maximum value always at location N-1



# BINARY SEARCH

- **The binary search algorithm can be used to search a sorted array for a specific value**
  - Look at middle element of sorted array
  - If equal to desired value, you found it
  - If less than desired value, search right half of array
  - If greater than desired value, search left half of array
  - Repeat until data is found (or no data left to search)

# BINARY SEARCH

- Search for value 7 in sorted array below


- Look at middle location (0+7)/2 = 3, which contains 5


- 5 < 7, so search to right


- This cuts size of array we are searching in half

# BINARY SEARCH

- Search for value 7 in unsearched array below


- Look at middle location (4+7)/2 = 5, which contains 7


- We found the desired value in only 2 searching steps!

# BINARY SEARCH

- Search for value 2 in sorted array below


- Look at middle location (0+7)/2 = 3, which contains 5


- 5 > 2, so search to left


- This cuts size of array we are searching in half

# BINARY SEARCH

- Search for value 2 in unsearched array below


- Look at middle location (0+2)/2 = 1, which contains 3


- 3 > 2, so search to left


- Now there is only one location to search!

# BINARY SEARCH

- Search for value 2 in unsearched array below


- Look at middle location (0+0)/2 = 0, which contains 2


- We found the desired value in only 3 searching steps!

# BINARY SEARCH

- **This divide and conquer approach is very fast since the array we are searching is cut in half at each step**
  - Consider an array with N=1024 sorted values
  - Searching we go from 1024 → 512 → 256 → 128 → 64 → 32 → 16 → 8 → 4 → 2 → 1
  - Only 10 steps needed to search array of 1024 elements

- **In general, binary search takes $\log_2 N$ steps to search a sorted array of N elements**
  - About 20 steps to search array of 1,000,000 elements
  - About 30 steps to search array of 1,000,000,000 elements

# BINARY SEARCH

- **To implement binary search we need to:**
  - Keep track of the portion of the array we are searching
  - Min = smallest array index of unsearched portion
  - Max = largest array index of unsearched portion
  - Mid = (Min + Max) / 2 is middle position
  - We need to initialize Min=0 and Max=N-1
  - We need to update these values as we search

- **This can be implemented using iteration or using recursion**

# BINARY SEARCH

```
// Iterative binary search
int Search(int Desired, int Data[ ], int Min, int Max)
{
  // Search array using divide and conquer approach
  int Mid = (Min + Max) / 2;
  while ((Data[Mid] != Desired) && (Max >= Min))
  {
    // Change min to search right half
    if (Data[Mid] < Desired)
      Min = Mid+1;
…
```

This loop will end when data is found or no locations are left to search

We change lower array index here to be 1 to right of midpoint

# BINARY SEARCH

```
...
    // Change max to search left half
    else if (Data[Mid] > Desired)
      Max = Mid-1;

    // Update mid location
    Mid = (Min + Max) / 2;
    System.out.println(Min + " " + Mid + " " + Max);
  }
...
```

We change upper array index here to be 1 to left of midpoint

# BINARY SEARCH

```
...
  // Return results of search
  if (Data[Mid] == Desired)
    return(Mid);
  else
    return(-1);
}
```

This returns the array index of desired data value or -1 if not found

# BINARY SEARCH

```
// Recursive binary search
int Search( int Desired, int Data[ ], int Min, int Max )
{
  // Terminating conditions
  int Mid = (Min + Max ) / 2;
  if (Max < Min)
    return(-1);
  else if (Data[Mid] == Desired)
    return(Mid);
…
```

This returns the array index of desired data value or -1 if not found

# BINARY SEARCH

```
...
  // Recursive call to search right half
  else if (Data[Mid] < Desired)
    return( Search( Desired, Data, Mid+1, Max ) );

  // Recursive call to search left half
  else if (Data[Mid] > Desired)
    return( Search( Desired, Data, Min, Mid-1 ) );
}
```

Notice how we search a smaller part of the array with each recursive function call

# ARRAY SORTING

- **The basic idea of array sorting is to move data values in the array so they are in numerical or alphabetical order**

- **There are many applications that need sorted arrays**
  - Output array in ascending or descending order
  - Search array more efficiently using binary search

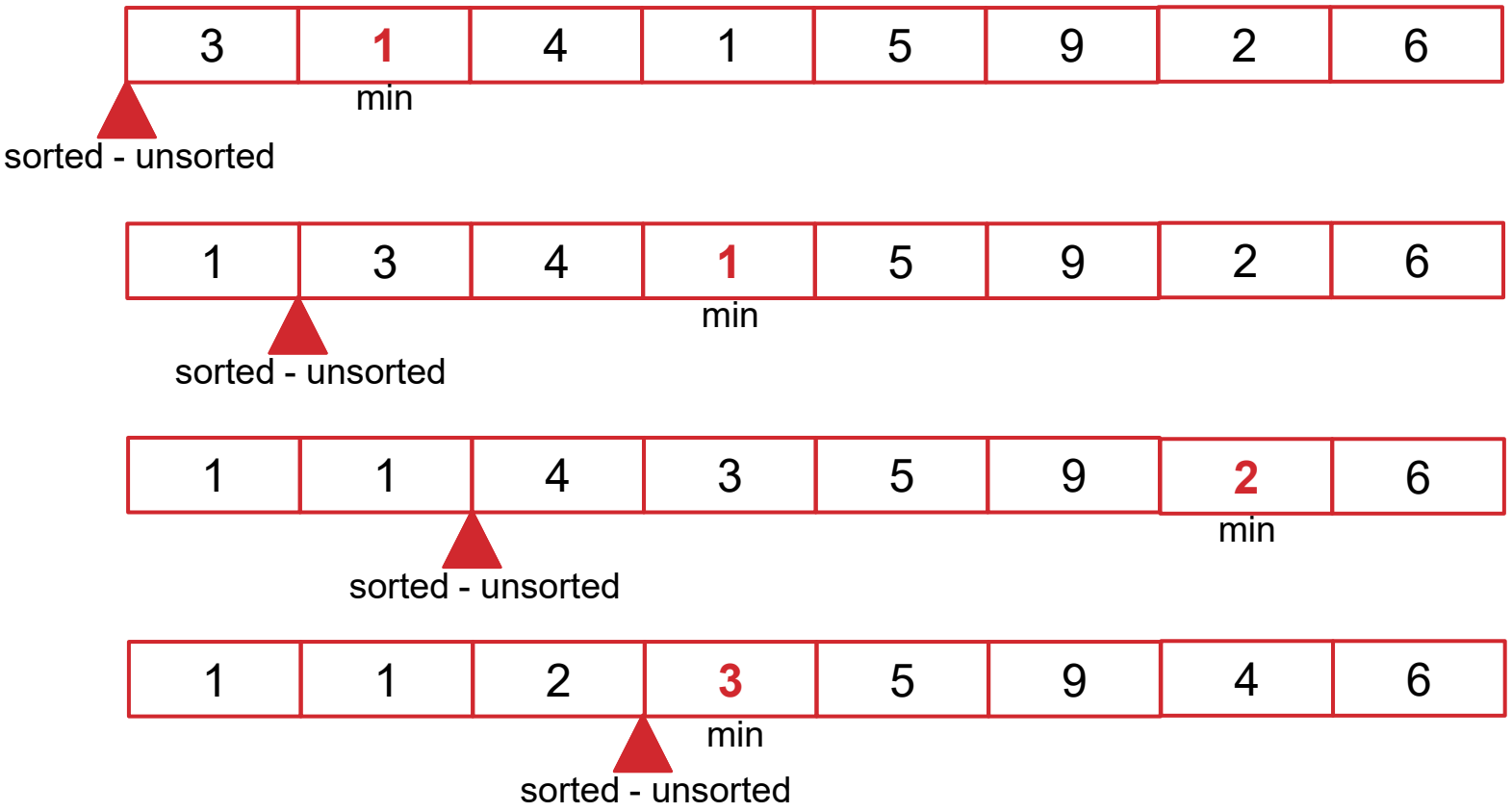
- **There are many algorithms for sorting arrays**
  - Some are easy to implement, others more complex
  - Some have fast run times, others are slower

# ARRAY SORTING

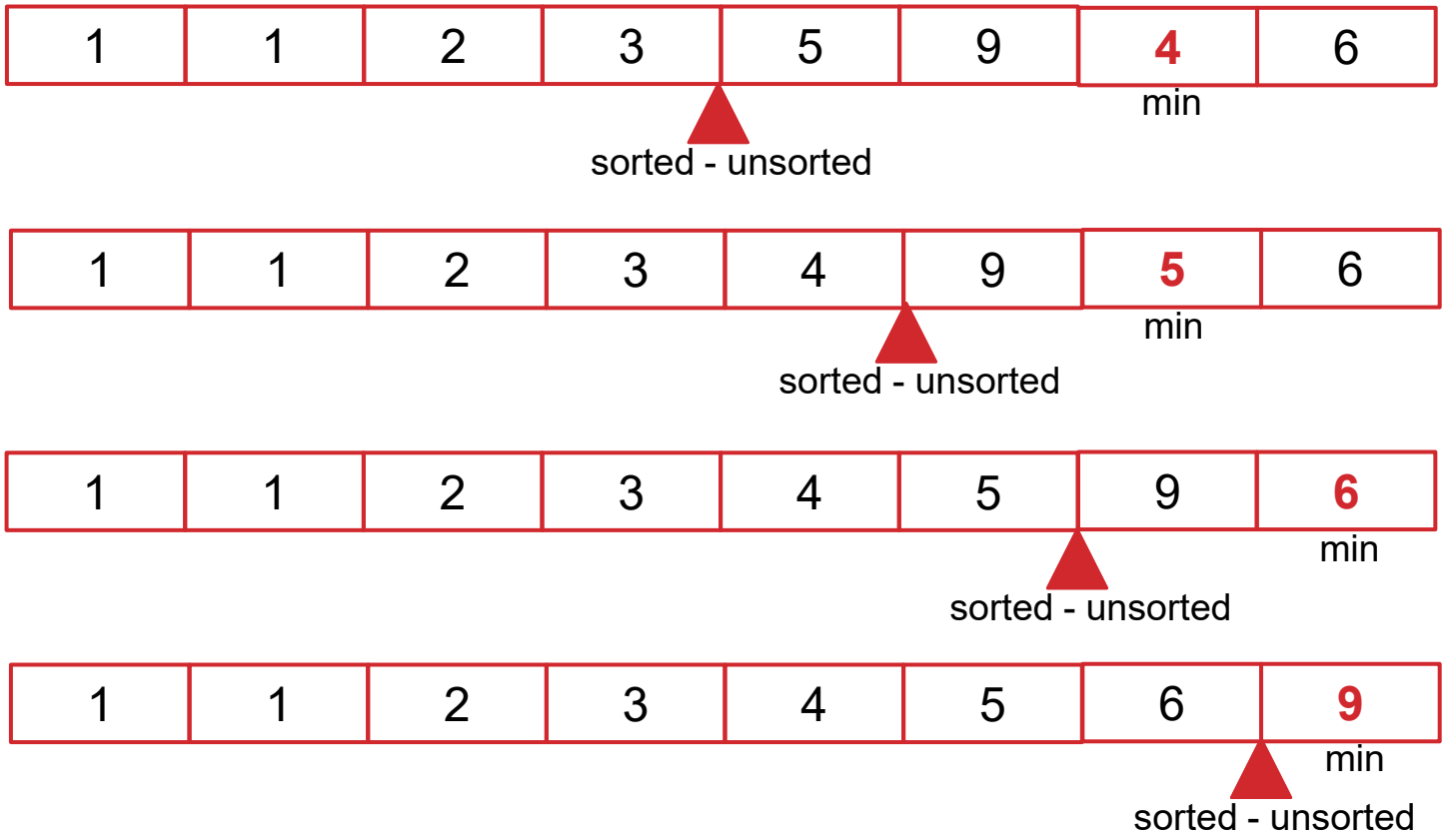
- **One easy algorithm to implement is selection sort**
  - Divide the array into two parts: sorted and unsorted
  - Find smallest value in the unsorted part of array
  - Swap value into end of sorted part of array
  - Repeat this process until the whole array is sorted

- **Consider an array containing the first 8 digits of PI**
  - Lets see what happens if we use selection sort
  - We show the array contents after each data swap

# ARRAY SORTING



# ARRAY SORTING



# ARRAY SORTING


- **The array is sorted when the unsorted part is empty!**
- **In general, selection sort will take N "find minimum and swap iterations" to sort an array of N elements**
- **Each "find minimum value" step looks at N data values, so selection sort takes $N^2$ operations**

# ARRAY SORTING

```
// Initialize data to sort
int Data[ ] = {3,1,4,1,5,9,2,6,5,3};

// Print unsorted data
for (int Index = 0; Index < Data.length; Index++)
   System.out.println(Index + " " + Data[Index]);
```

# ARRAY SORTING

```
// Perform selection sort algorithm
for (int Index = 0; Index < Data.length; Index++)
{
  // Find smallest value in unsorted part
  int SmallPos = Index;
  for (int Pos = Index; Pos < Data.length; Pos++)
    if (Data[Pos] < Data[SmallPos])
      SmallPos = Pos;
  …
```

This loop executes N times moving the sorted-unsorted line

This loop executes N times to find the smallest data value

Notice that we start this loop at the beginning of the unsorted part of array

# ARRAY SORTING

```
   …
   // Swap smallest value into sorted part
   int SmallVal = Data[SmallPos];
   Data[SmallPos] = Data[Index];
   Data[Index] = SmallVal;
}

// Print sorted data
for (int Index = 0; Index < Data.length; Index++)
   System.out.println(Index + " " + Data[Index]);
```

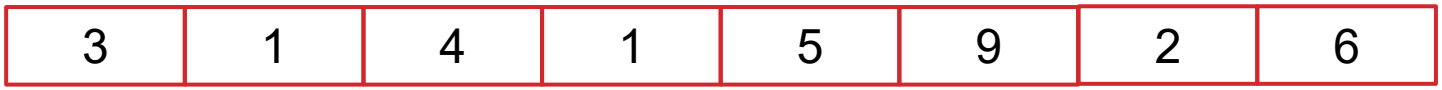
# CALCULATING MEDIAN VALUE

- **The median is defined to be midpoint of set of values**
  - Half of the data values are larger
  - Half of the data values are smaller

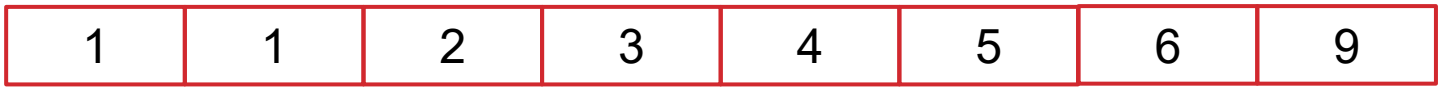
- **Algorithm for calculating the median value**
  - Sort data into numerical order
  - Calculate midpoint = array_size / 2
  - Median value = data[midpoint]

- **Calculating the median is more work than finding the average, but it is considered to be a more robust statistic**

# MEDIAN VALUE




Unsorted array




Sorted array

midpoint = Data.length/2 = 4 so
the median value is Data[4] = 4

There are 4 values < median
and 3 values > median because
we the array length is even

# CODE DEMO

**LinearSearch.java**

**SelectionSort.java**

**BinarySearch.java**

# ARRAYS LIBRARY

- **The developers of Java have created a very powerful library of array manipulation functions in the Arrays class**

- **This class has functions in several categories**
  - Copying functions
  - Comparison functions
  - Sorting functions
  - Searching functions
  - Utility functions

- **To use this library you must import java.util.Arrays**

# ARRAYS LIBRARY

- **Copying functions include:**
  - static float[] **copyOf**(float[] array, int length) – copies the specified array, truncating or padding with zeros as necessary so the copy has the specified length.
  - static float[] **copyOfRange**(float[] array, int from, int to) – copies the specified range of the array into a new array.

  - static void **fill**(float[] array, float value) – assigns the float value to all elements in array.
  - static void **fill**(float[] array, int from, int to, float value) – assigns the float value to specified range in array.

# ARRAYS LIBRARY

- **Comparison functions include:**
  - static boolean **equals**(int[] a1, int[] a2) – returns true if the two specified arrays of are equal to one another.
  - static boolean **equals**(int[] a1, int from1, int to1, int[] a2, int from2, int to2) – returns true if the two specified arrays of are equal to one another over the specified ranges.
  - static int **mismatch**(int[] a1, int[] a2) – finds and returns the index of the first mismatch between two arrays, otherwise return -1 if no mismatch is found.
  - static int **mismatch**(int[] a1 int from1, int to1, int[] a2, int from2, int to2) – finds and returns the index of the first mismatch over specified ranges.

# ARRAYS LIBRARY

- **Sorting functions include:**
  - static void **sort**(int[] array) – sorts the array in ascending numerical order.
  - static void **sort**(int[] array, int from, int to) – sorts the specified range of the array in ascending numerical order.

  - static void **parallelSort**(int[] array) – sorts the array in ascending numerical order using a parallel algorithm.
  - static void **parallelSort**(int[] array, int from, int to) – sorts the specified range of the array in ascending numerical order using a parallel algorithm.

# ARRAYS LIBRARY

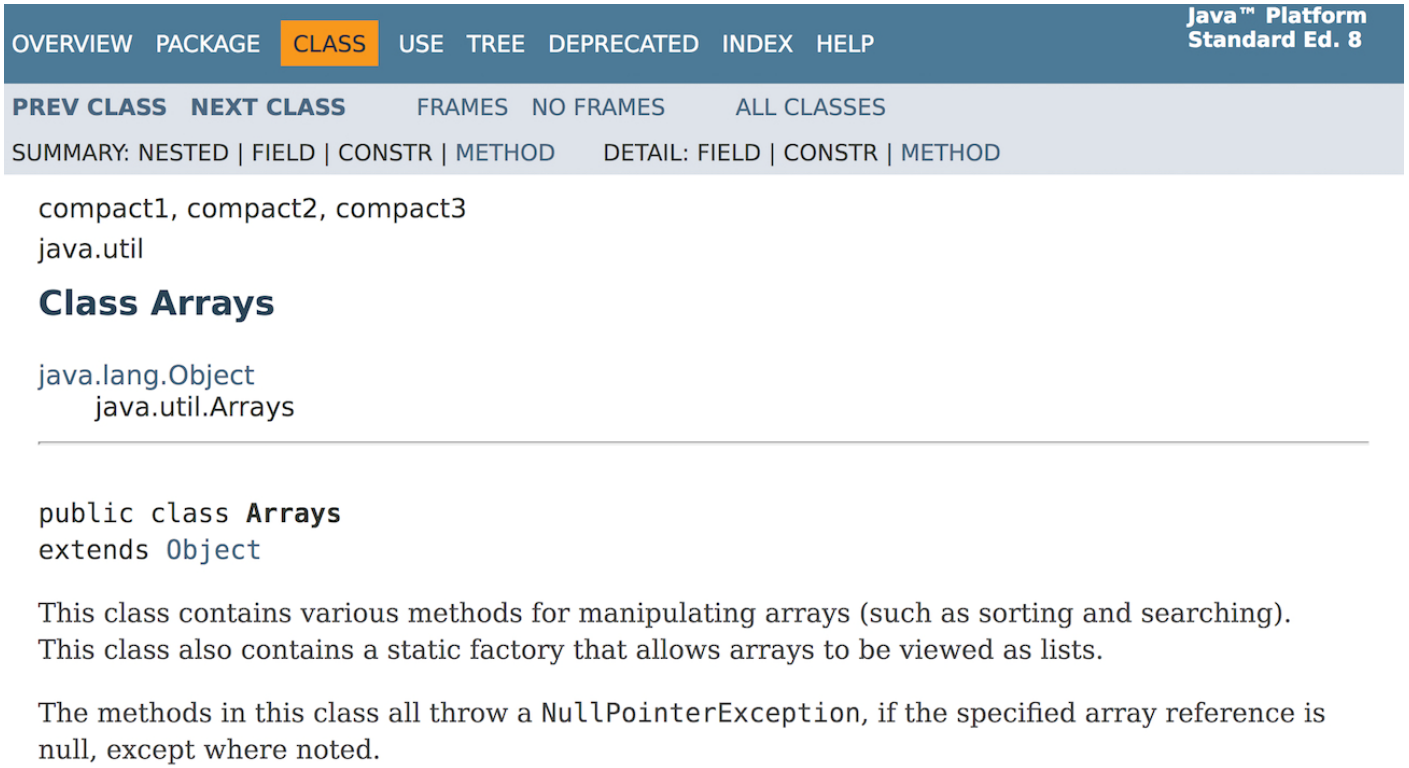
- **Searching functions include:**
  - static int **binarySearch**(float[] array, float value) – searches a sorted array for the specified value and returns its array location (or returns -1 if the value is not found).
  - static int **binarySearch**(float[] array, int from, into to, float value) – searches a sorted array in specified range for the specified value and returns its array location (or returns -1 if the value is not found).

# ARRAYS LIBRARY

- **Utility functions include:**
  - static int **hashCode**(int[] array) – returns a hash code based on the contents of the specified array. For any two arrays a and b such that Arrays.equals(a, b), it is also the case that Arrays.hashCode(a) == Arrays.hashCode(b).
  - static String **toString**(int[] array) – returns a string representation of the contents of the specified array. This makes it possible to print an entire array using one print command.
    - System.out.println(toString(num));

# ARRAYS LIBRARY

- **See the official Java documentation here**



OVERVIEW PACKAGE CLASS USE TREE DEPRECATED INDEX HELP — Java™ Platform Standard Ed. 8

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3

java.util

## Class Arrays

java.lang.Object
    java.util.Arrays

---

```
public class Arrays
extends Object
```

This class contains various methods for manipulating arrays (such as sorting and searching). This class also contains a static factory that allows arrays to be viewed as lists.

The methods in this class all throw a `NullPointerException`, if the specified array reference is null, except where noted.

# SOFTWARE ENGINEERING TIPS

- **Suggestions when using arrays:**
  - Use the .length attribute for the array dimensions
  - Make sure your array indices are within 0..N-1
  - Use functions to implement useful array operations

- **Common programming errors:**
  - Invalid array declarations or initializations
  - Array index out of bounds (off by one errors)
  - Missing [ ] in array parameter definitions

# SUMMARY

- In this section, we described how linear search can be used to find the min/max or special values in an array
- Then we described how “binary search” can be used to very quickly search for values in a sorted array
- Next we introduced the “selection sort” algorithm and showed how it can be used to sort data
- We also saw how a sorted array can be used to calculate the median value
- Finally, we introduced the functions in the Math and Arrays libraries