

# Java Arrays

- Java Arrays
- Java References and Equality
- Enhanced `for` Loops.

# Strings and Characters

- We have seen that each string has zero or more characters
  - String hello = “hello world” has 11 characters
- Characters in a string are indexed beginning with index 0
- Strings have a length given by `String.length()`
  - e.g. `hello.length()`
- The last valid index is `String.length() - 1`

# Strings, Characters, and Sequences

- We can say that a string is a sequence of characters
- in a sequence, order matters: “abc” is different from “bca”, “cab”, “acb”, “bac”, and “cba”
- in a string, this sequence is immutable: we cannot change part of a string
  - but we can build a new string, e.g. by concatenation, substring, or any combination of these
- we may have a use for a sequence of integers, e.g. 1, 2, 3
- or doubles, e.g. 3.14, 2.718, 1.41
- just as we can loop over the characters of a string, we may want to loop over the elements of such a sequence
- and it might be nice if the sequence is mutable

# Java Arrays

- An array is a mutable sequence of values of a given type
  - every element of the array has the same type, e.g. int, double, String, boolean, ...
  - we can change the value at a given index
  - we cannot change the length of the array
- Arrays are declared using a square bracket notation, and initialized with the keyword `new`

```
int myNumbers[] = new int[10];  
String myNames[] = new String[3];
```

- Method parameters can be arrays:

```
public static void main(String[] arguments) { ...
```

# Array Initialization

- If you know the values that you want to have in an array when you declare it, you can specify them in the initialization:

```
double mathConstants[] = { 3.14, 2.718,  
1.41 };
```

- in this case, `new` is not needed

- when `new` is used, the array values are automatically initialized to a default value, e.g. 0 for an `int` array, 0.0 for an array of doubles

# Array Elements

- Array elements can be used like any variable:

```
public static void main(String[] a) {  
    if (a.length > 0) {  
        System.out.println("first arg: " + a[0]);  
    }  
}
```

- this means we can also assign to them:

```
int numbers[] = new int [100];  
numbers[0] = 17;  
numbers[99] = 32;
```

- and, like any variable, use the value in expressions  
if (numbers[7] > numbers[6]) ...  
numbers[i] = numbers[i-1] + 10;
- The first index is 0, just as for characters in strings

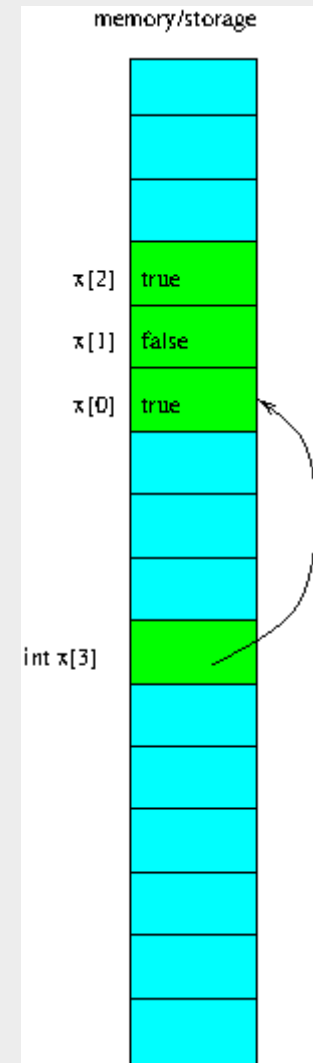
# Looping over Array Elements

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

- the first index is 0
- the last index is `a.length - 1`
  - this is the same as for the index of characters in a string
- so an array of 5 elements has elements at indices 0, 1, 2, 3, and 4

# Array Storage in Memory

- A declaration such as  
`boolean x[] = new boolean[3];`  
does three things:
- it **allocates** (reserves) **storage** (memory) for an array of 3 boolean values
- it allocates storage for a variable `x`
- and has `x` **refer** to storage for the array
  - the parts of memory that have been allocated (reserved) for this example are shown in green
- really, `x` is a kind of number (a **pointer**) that refers to the first location in memory where the array values are stored



# Array References

- Because an array variable stores a reference, two variables can refer to the same underlying array:

```
int a1[] = { 1, 2, 3, 4 };
```

```
int a2[] = a1;
```

```
a2 [3] = 55;
```

```
if (a1[3] == 55) { ...
```

- the condition of this `if` is true!
- so an assignment of an array reference is different from copying the array

# Array Copy

- When we want to copy an array, we do so explicitly:

```
int a1[] = { 1, 2, 3, 4 };  
int a2[] = Arrays.copyOf(a1, a1.length);  
a2 [3] = 55;  
if (a1[3] == 55) { ...
```

- now, the condition of this `if` is false
- `Array.copyOf` allocates new space in memory for the copy of the array, and returns the reference (pointer) to the new array

# Arrays.copyOf

- `Arrays.copyOf` takes two parameters: an array `a` and a length `n`
- if `n == a.length`, the entire array is copied
- if `n < a.length`, the first `n` elements of `a` are copied
- if `n > a.length`, an array of size `n` is created, and the first `a.length` locations are copied from `a`
  - the remaining locations are initialized with a default value, e.g. 0 for int arrays

# Understanding Arrays.copyOf

- the result of `Arrays.copyOf` is of a type that depends on the type of the argument

- for now we will write this as *someType*

```
public static someType[]  
    copyOf(someType[] a, int n) {  
    someType result[] = new someType[n];  
    for (int i = 0; i < a.length && i < n;  
        i++) {  
        result[i] = a[i];  
    }  
    return result;  
}
```

- `new` allocates storage for the new array and initializes all the elements to the default value
- the loop re-initializes the first `n`, or `a.length`, elements of the new array

# More about References

- Just like an array variable, a String variable is a reference to memory allocated to store the characters of the string
  - and the length
- Because they are references, `s1 == s2` compares whether `s1` and `s2` refer to the same underlying memory
- and `String.equals` is used to compare the contents of the strings: the lengths and the actual characters
- in Java, both arrays and Strings are Objects
- All Object values in Java are references to the memory allocated to store the values of the object
- So the `==` comparison tells us whether two objects are the very same, that is, whether they refer to the same memory, while the `Object.equals` method **may** give us a more meaningful comparison
  - *may*, because comparing object contents is not always meaningful

# Partially Filled Arrays

- Arrays are fixed size
- If the data we want to store in the array has size  $n$  (where  $n$  is not fixed), we can:
  - allocate an array as large as the largest possible value of  $n$
  - at any given time, only use the first  $n$  elements of the array
- we usually have a variable to keep track of the current value of  $n$
- since every array location has a value, array locations  $n$  through  $a.length-1$  store values, but these are values we do not use

# Partially Filled Arrays: Example

```
int numLines = 0;
final int MAX_LINES = 1000;
String lines[] = new String[MAX_LINES];
while (in.hasNext() && numLines < MAX_LINES) {
    lines[numLines++] = in.next();
}
```

- at the end, linesFromUser has the number of lines the user entered, and lines[0..linesFromUser-1] has the actual lines
- we don't worry about the remaining elements
- when declaring arrays of fixed size, it is a good idea to use a constant as the fixed size
- this constant can be used in the loop condition

# Bounds Errors and Buffer Overflow

- In the previous example, suppose MAX\_LINES was a small number, and we didn't test for it in the loop
- eventually, we might try to store a value in lines[numLines] when numLines  $\geq$  lines.length
- this is an error
- in Java, such an error throws an exception
  - in other languages, it may overwrite memory unrelated to the array: a **buffer overflow**
  - in such languages, buffer overflow is hard to detect and may cause serious problems
  - see Random Fact 6.1 in the textbook
- in Java, negative indices also throw exceptions

# Enhanced For Loop

- It is very common to want to loop over all the elements of the array
- In the special case where we:
  - are only reading these elements (not assigning to them), and
  - don't need the loop index,

there is a special syntax, called the **enhanced for** loop

- again, we use *someType* to stand in for the type of the array elements:

```
for (someType e: a) {  
    // inside the loop, e is a local variable  
    // assigning to e does not change the array element!  
}
```

- If the array has 0 elements, the body of the loop is never executed

# Enhanced For Loop: Syntax

- `for ( type-of-the-array-element name-of-local-variable-holding-the-element : array ) {`
- the type of the array element must match the types of the elements of the array
- the name of the local variable is up to the programmer
- the array is usually a variable, but could be an expression such as the result of a call to `Arrays.copyOf`
- The parentheses and colon are required
- The index of the element is not available in the body of the loop -- if it is needed, use a conventional for loop with an explicit index

# Example:

## Finding the Maximum Value

- Suppose we are given an array of double numbers, and want to find the largest

```
double a[] = ...
```

```
double max = a[0];
```

```
for (double element: a) {
```

```
    if (element > max) {
```

```
        max = element;
```

```
    }
```

```
}
```

- at the end of the loop, max has the value of the largest element of the array

# Summary

- Arrays are objects that can hold a fixed number of values of a given type in a specific sequence
- Arrays are mutable: assigning to an array element changes the value in the array  
`a[0] = "hello world";`
- The first index is always 0
- `array.length` is the number of values, and `array.length - 1` is the last index
- `new` allocates memory to hold the values of an array (or any object)
- `Arrays.copyOf` copies contents of arrays (or we can write a for loop)
- The enhanced for loop creates a new variable for each array element and makes it available inside the loop