

# Review Java strings: The String type

```
String fName ;  
fName = "William ";
```

```
String lName = "Sakas";
```

```
String mName = new String("Gregory ");
```

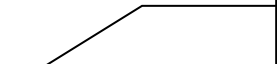
```
System.out.println(fName +mName+lName);
```

```
int gregNumOfChars = mName.length();
```

```
System.out.println(gregNumOfChars); // 7
```



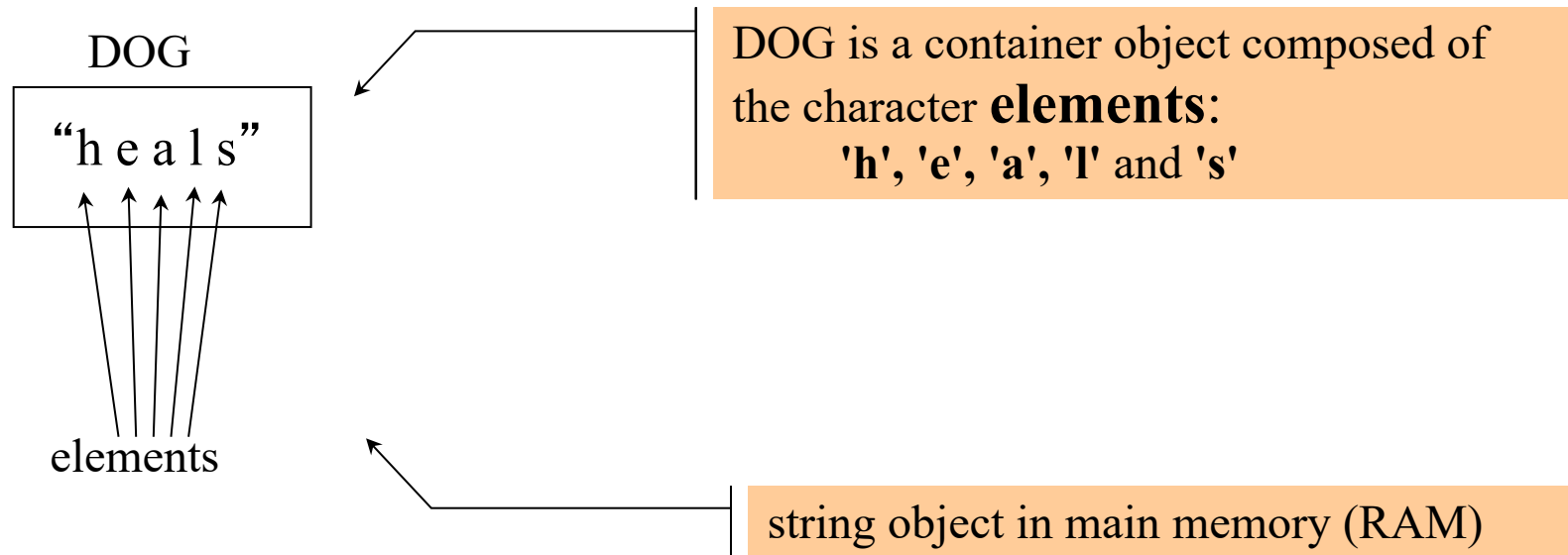
Three ways to create an initialize strings



There are also useful string functions, note the “dot” – the length function is a string METHOD – much more on this later.

The String is a *container* type - Strings contains a collection of sub-*elements* which can have different values

String DOG = "heals" ;



# Additional functionality provided by the string type:

## Read “**in position**” access

DOG

h	e	a	l	s
0	1	2	3	4

string objects are made up of char **elements** and each element has an **index** starting from 0.

in position READ access to element with index 2

```
System.out.println( DOG.charAt( 2));
```

**charAt** member function is an *indexing or subscripting* function  
In your head, interpret charAt as “in position” or “at position”  
“**output DOG IN POSITION 2.**”  
“**output DOG AT POSITION 2.**”

\* experienced programmers often use "sub" when reading off the subscript operator.

DOG

h	e	e	l	s
0	1	2	3	4

Integer variables are often used to hold indices.

```
int anInt = 1 ;  
System.out.println( DOG. charAt (anInt) );  
anInt++ ;  
System.out.println( DOG.charAt ( anInt ) );  
anInt++ ;  
System.out.println( DOG.charAt ( anInt ) );
```

**TASK:** Write some C++ code to output a string in reverse order.

precondition: aString == "heels"

postcondition: **sleeh** is displayed on the screen.

Hint: Use an integer as an index. Which index do we start with?

# Java Arrays

## Like Strings –

- **Arrays** contain a list of elements
- **Arrays** have in position read access but using [ ] brackets.
- You can access the **length** of an **Array** but with the instance variable, **.length** not the method **.length()**.

## Arrays differ from Strings::

- *Most important:* **Arrays** can hold any type of data, not just strings.
- **Arrays** don't have the operators < , ==, >>, +.
- **Arrays** are **mutable** unlike Strings — the content of an Array can be changed while a program is running:
  - there is "in position" **WRITE** access.
- One can't use a Scanner object to input an entire Array at one.

# Java Syntax for arrays:

To declare an array:

```
int theArray[]; // this creates a variable called "theArray"  
                // that can reference an array of integers.
```

```
theArray[] = new int[17]; // this creates an actual array  
                        // that can contain 17 integers  
                        // and be referenced by theArray.
```

**Note:** When first created, the elements in the arrays contain a default value, for integers it is 0, for doubles it is 0.0, etc.. But you can initialize an array when it is declared:

```
int theArray[] = new int[] {55, 65, 70, 75, 85};
```

```
// theArray is of size 5 and contains 55, 65, 70, 75, 85
```

# Additional functionality provided by the string type:

Read “**in position**” access

DOG				
h	e	a	l	s
0	1	2	3	4

String DOG = "heals"

System.out.println( DOG.charAt(2));

anArray				
55	65	70	75	85
0	1	2	3	4

int theArray = new int [] {55,65, 70, 75, 85};

System.out.println( anArray[2] );

[ ] is an *indexing or subscripting* operator  
In your head, interpret [ ] “in position” or "at position"  
“**output DOG IN POSITION 2.**”  
“**output DOG AT POSITION 2.**”

\* experienced programmers often use "sub" when reading off the subscript operator.

# Additional functionality provided by arrays, NOT like Strings:

“**in position**” write access

DOG				
h	e	a	l	s
0	1	2	3	4

String DOG = "heals"

What if we want to change  
"heals" to "heels"?

Well in Java, strings are  
**immutable**, which means their  
contents can't be changed.

So, there is no method to change  
one character in a string. You would  
have to overwrite the whole string:

DOG = "heels";

anArray				
55	65	70	75	85
0	1	2	3	4

```
int theArray = new int [] {55,65, 70, 75, 85};
```

Arrays are **mutable** — they can be  
changed. The **[ ]** operator provides in  
position **write access** as well as read  
access.

```
anArray[2] = 90; // overwrites the 70  
                // with the integer 90
```

anArray				
55	65	90	75	85
0	1	2	3	4



# Java Arrays More examples

```
String stringArray[];  
stringArray = new String[4];
```

```
stringArray[1] = "Juli" ;  
stringArray[2] = "Jorge";  
stringArray[3] = "Joel" ;
```

## NOTE:

- The length of stringArray is 4, not 3
- stringArray[0] == "" // i.e, the null string
- This is different than:

```
String stringArray[ ] = new String[ ] {"Juli", "Jorge", "Joel"}
```

- Why?

# Java Arrays:

When the programmer can determine the exact size of the array

```
Scanner input = new Scanner(System.in);
```

```
System.out.print ("How many close friends do you have?");  
int n = input.nextInt();
```

```
input.nextLine(); // clear newline  
String myFriends[] = new String[n];
```

```
for (int i; i < myFriends.length; i++){  
    System.out.print("Enter a name of one of your friends: ");  
    myFriends[i] = input.nextLine();  
}
```

Integer variable to hold array size .

## Java Arrays:

When the programmer can't predict the size of the array

Much more common!!!

It's often the case that you don't know how big your data is, or you don't want to burden the user by declaring a size.

Enter a name of one of your friends. Type "Stop" to end.

Overestimate how many friends someone might type in before typing "Stop"

```
String myFriends[] = new String[1000];
```

← Max array size .

So, you might have the first three elements filled with names, but the other 997 elements filled with the null string.

You, the programmer has the responsibility to keep track of the used or inhabited size of the array.

```
int numOfFriends = 0;
```

← Programmer codes to keep track of actual size.

```
String myFriends[] = new String[1000];
```

Max array size .

```
int numOfFriends = 0;
```

```
String aFriend = "" ;
```

```
while ( ! aFriend.equals("Stop") ) {
```

```
    System.out.print(
```

```
        "Enter a friend's name. Type Stop to end:");
```

```
    aFriend = input.nextLine();
```

```
    myFriends[numOfFriends] = aFriend;
```

```
    numOfFriends++;
```

```
}
```

Actual Size based on  
how many names were  
entered

```
System.out.println("You have "+numOfFriends+" friends!");
```

```
for (int i=0; i<numOfFriends; i++)
```

```
{
```

```
    System.out.println(myFriends[i]);
```

```
}
```

This method is used 99% of the time in the real world.

# Java Arrays

Its really really really important for you as a human programmer to always keep track of your arraysize – assuming the array is **"partially filled"**.

For example, when you delete an element from an array, it's **your** responsibility to do:

```
theArraySize--- ;
```

after you shuffle down all the elements.

## Some standard algorithms for arrays.

1. display the elements in an array
2. get elements elements from a user to fill an array
3. Add a single element to an array
4. find the first instance of an element in a array (sequential search)
5. delete the nth element in a array

**IMPORTANT:**For the rest of the examples in these slides, assume that there is an array of ints declared called **theArray**. And the current Size of the array is stored in **currSize**, and that there a declared Scanner object called **input**.

That is, assume that:

```
int MAX_SIZE = 1000;  
int theArray[ ] = new int[MAX_SIZE] ;  
int currSize = 0 ;
```

is in the same scope (or block) that the example code is in.

BTW, normally this would be a really bad name! A good programmer would call the array a name that helps identify what type of elements the array contains (e.g. **theIntegers**, **theCards**, **LastNames**, etc.)

## 1) Display all the elements of a array

*// NOTE: "i" is a very, very common shorthand for "the current Index"*

```
for (int i=0; i < currSize; i++ )
{
    System.out.print(theArray [ i ] +" ");
}
System.out.println( );
```

## 2) Get elements from the user to fill an array (There's a problem ...)

```
int currSize = 0;
int anInt = 0;

while ( anInt != -9999 ) { // -9999 adhoc signal to stop
    System.out.print("Enter an integer. Type -9999 to stop:");

    x = input.nextInt();
    theArray[currSize] = x

    currSize++;
}
```

### 3) Add an element to an array (Same problem here.)

```
int aNewElement;
```

```
System.out.print("What integer would you like to add to the array? ");
```

```
aNewElement = input.nextInt();
```

```
theArray[ currSize ] = aNewElement ;
```

```
currSize++    ;
```



#### 4) Find the first instance of an element in a array.

*// remember i is a common shorthand for "the current Index"*

```
int i = 0 ;
```

```
int theTargetItem = input.nextInt();
```

*// the order of the && comparisons here are CRUCIAL! Why?*

```
while ( i < currSize && theArray [ i ] != theTargetItem )
```

```
{
```

```
    i++ ;
```

```
}
```

*// postcondition of the while loop*

*// Either:*

*// i) we've found the target (the loop ends when*

*// theArray[i] == theTargetItem so i < the size of the array) or*

*// ii) we've gone past the end of the array (i == Size of the array )*

```
if ( i < currSize ) // we've found the target
```

```
{
```

```
    System.out.println( "The number you entered is in position " ) ;
```

```
}
```

```
else // i is equal to currSize and the target has not been found
```

```
{
```

```
    System.out.println( "The number is not in the array. " ) ;
```

```
}
```

## 5) Delete the $n^{\text{th}}$ element of a array

```
int targetPosition;  
cout << "Enter the position of the number you want to delete: ";  
cin >> targetPosition ;
```

*// "shuffle down" the elements. First by overwriting the element in the targetPosition with  
// whatever was in theArray in the targetPosition+1 and then continuing to the end  
// of the array. Question: Why is the go again condition size()-1 and not simply size() ?*

```
for (int i=targetPosition; i < currSize-1; i++)  
{  
    theArray[ i ] = theArray[ i+1 ] ; // this is easy to think about if you read it as  
                                     // "English" in your head:" theArray in position i  
                                     // becomes theArray in position i+1"  
}
```

*// Not quite done. The element in the targetPosition has been overwritten, but by  
// shuffling down all the elements, we're left with two identical elements at the end  
// of the array. (the last element "shuffled down" to overwrite the element in the  
// next to last position, but nothing was down with the element in the last position.  
// No problem, just pop it off the end of the array:*

```
currSize = currSize - 1
```