Strings in Java – String Objects

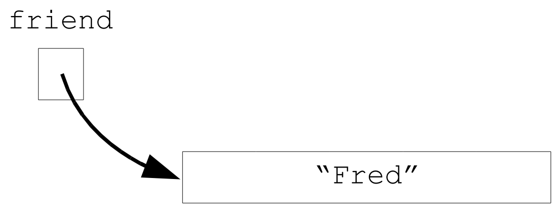
Strings in Java are objects. They are instances of the class `String` (in the package `java.lang`).

As is the case with other objects, `String` variables are actually references to a `String` object in memory. The variable with contain an address, rather than the actual string.

For example, the statement

```
String friend = new String(“Fred”);
```

creates a new string as shown in the following diagram.

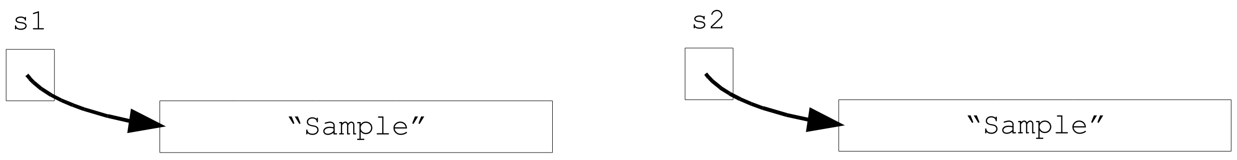


This constructor is rarely used. Java has an equivalent, simpler form to create and initialize a variable of type `String`.

```
String friend = “Fred”;
```

The constructor can be useful, however, to create multiple string objects with the same value (but different instances of each object).

```
String s1 = “Sample”;
String s2 = new String(s1);
```



It is also possible for a String variable to be in a state where it does not refer to a string containing characters. Consider the following:

```
String s1;            // uninitialized
String s2 = null;     // null string
String s3 = "";       // empty string
```

The variable `s1` is uninitialized. It can be assigned to a String, but any other attempt to use this variable will result in an error.

The variable `s2` has been set to the `null` value. It can be printed (producing the output "null") or compared to other strings.

The variable `s3` has been set to the empty string, which has a length of zero and contains no characters, but is also valid to use with all string operations.

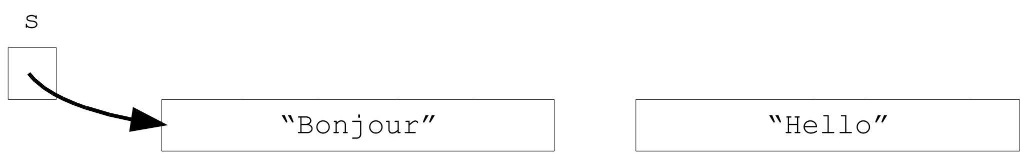
Changing a String Value

The String class contains no *mutator* methods. Once we create a string, its value cannot be changed. Because of this, we say strings are *immutable* objects. We can still change the values of the string variable, to reference another string, but the actual string object (in memory) is fixed.

For example, consider the fragment

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

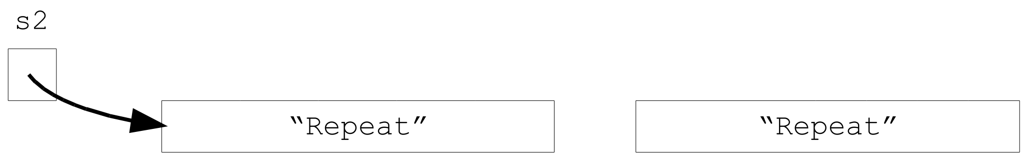
The first line sets the variable `s` to refer to a string object containing `"Hello"`. The second line changes the reference to another object containing `"Bonjour"`. The first object, `"Hello"`, which now has no variables referencing it, it now lost.



Now consider the fragment

```
String s2 = "Repeat";
s2 = "Repeat";
```

Once again, the first line creates s2 and sets it to refer to a string object. The second line creates a *new string object*, even though the strings are exactly the same. The result is the same – two string objects are in memory, but the variable references the second object, and the first has been lost.



Exercises

1. What would be the output of the following fragment?

```
String s;
System.out.println(s);
```

2. What would be printed by this fragment?

```
String s, t, u;
s = null;
t = "";
u = " ";
System.out.println("s is |" + s + "|");
System.out.println("t is |" + t + "|");
System.out.println("u is |" + u + "|");
```

3. What would be printed this fragment?

```
String s = "Sample";
String t = s;
s = "Simple";
System.out.println("s is: " + s);
System.out.println("t is: " + t);
```

4. What would be printed this fragment?

```
String s = "First";
String t = new String(s);
s = "Second";
System.out.println("s is: " + s);
System.out.println("t is: " + t);
```

5. What do we mean when we say strings are *immutable*?

6. Is the following fragment legal? Explain.

```
String s = "old";
s = "new";
```

Many problems involve the manipulation of large amounts of data – for example, lists and tables of data. For many such problems, an array provides an appropriate structure for storing and organizing the data.
Consider the following table, which contains price information for one litre of gasoline from 1980 to 2000:

Year	1980	1985	1990	1995	2000
Price ($)	0.27	0.51	0.59	0.56	0.72

To keep track of these data in a computer program, we might use variables: price1980, price1985, price1990, price1995, and price2000. Although valid, this strategy is cumbersome, and would become even more inconvenient if we wanted to add additional prices.
In contrast, Java offers a data structure called an *array*, which is a collection of data items of the *same type*. In this example, we could store our gasoline prices in an array of 5 *elements*. We might name the identifier for the array `price`, and each element of the array would be referenced using an *index*.
In Java, every array index starts at zero (0).

	0	1	2	3	4
`price`	0.27	0.51	0.59	0.56	0.72

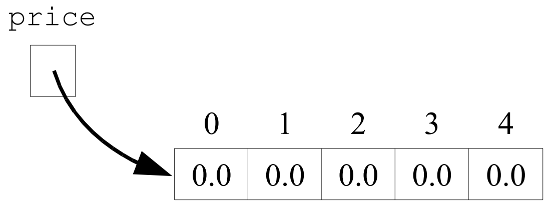
The identifiers of each element in the array are `price[0]`, `price[1]`, and so on.
In Java, an array is a type of object. To create an array, we follow the same steps used for creating any other object, and we must consider, and avoid, the same possible sources of errors.
To create an array to hold the gasoline price data, we could declare

```
double[] price;
```

This create the variable `price` and determines that `price` is a reference to an array of `double` values. As with other objects, however, this does not actually create the array. It simply creates the variable that can act as a *reference* to the array once it has been created.
To actually create the array in memory, we use the new keyword:

```
price = new double[5];
```

which creates the following pictorial representation in memory.



Notice the value 0.0 in each element of the array. Numeric arrays are automatically initialized to zero when space is allocated to them. For `char` values, they are initialized to the *null character*. For `boolean` arrays, they are initialized to `false`.
This same declaration, creation, and initialization can be expressed as a single command:

```
double[] price = new double[5];
```

Length of an Array

Once an array has been created in memory, its size is fixed for the duration of its existence. Every array object has a `length` field whose value can be obtained by appending `.length` to the array identifier.

```
int len = price.length;
```

Since all arrays start numbering at zero, the length is always one greater than the highest index.

Explicit Array Declaration and Initialization

Java has another form of array declaration that allows us to initialize the array with any values desired at the time of declaration.

```
int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29};
```

Notice that we have not used the keyword `new`, and we have used brace brackets instead of square brackets. The size of the array is not stated, but it is understood to match the number of elements included in the array declaration.

Traversing Arrays

To access each element in an array, a counted, or `for` loop, is traditionally used. Suppose we have an array of 100 boolean values, automatically initialized to false. To change all of these to true, we could write

```
boolean[] flags = new boolean[100];
for (int i = 0; i < flags.length; i++)
      flags[i] = true;
```

Note that we have used the `flags.length` to determine the upper bound of the loop. As we previously noted, the `.length` field will return a value one greater than the maximum index of the array, which is why our loop must use `i < flags.length` to avoid running past the end of the array. If we attempt to use an index that is outside of the range of the array, Java is throw an *exception*.

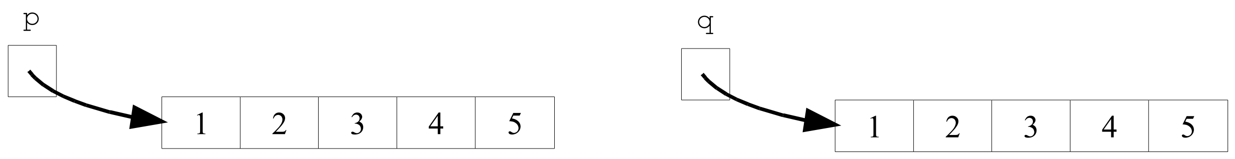
Comparing Arrays

Arrays are objects, and as such, they have the same limitations we experience with other objects. One of those involves the notion of equality.

The array variable is a *reference* to an array object. In other words, it points to a location in memory which holds the array and its data. We must keep this in mind when using the assignment statement (=) and the equality operator (==).

For example, suppose we have made the following declaration:

```
int[] p = {1, 2, 3, 4, 5};
int[] q = {1, 2, 3, 4, 5};
```



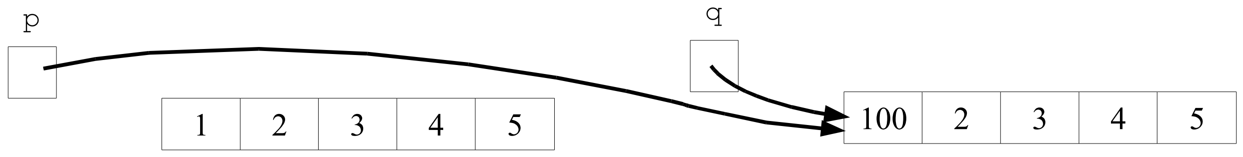
Though common sense , it would be reasonable to say they arrays are equal. They contain the same number and type of elements, and each corresponding element contains the same value. Nonetheless, if we were to test their equality,

```
System.out.println(p == q);      // false
```

The result would be `false`. The reason is that the variables `p` and `q` are *references* to memory locations, or addresses. Since each array is distinct in memory, it must have a unique address, so they are definitely not equal.
Note that Java contains a method, `Arrays.equals`, that compares the equality of the content of arrays.
Similarly, using the assignment operator (=) could produce an unexpected situation.

```
p = q;
q[0] = 100;
System.out.println(p[0]);  // outputs 100
```

In this case, the variables p and q now point to the same memory location, and the data contained in the original p array is forever lost.



Arrays as Method Parameters

Like other objects, arrays can be parameters of methods, and can be returned by methods. Since the array variable is a *reference*, the method will make a copy of the reference, so the original cannot be changed by the method. The *referenced data*, however, can and will be changed by the method unless you keep this in mind (and sometimes you want to change array data using a method, so this is desirable).

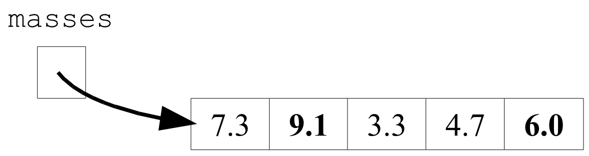
```
public static double[] join (double[] a, double[] b)
{
      double[] result = new double [a.length + b.length];
      int i, j;
      for (i = 0; i < a.length; i++)
            result[i] = a[i];
      for (j = 0; j < b.length; i++, j++)
            result[i] = b[j];
      return result;
}
```

Note that when introducing arrays, we discussed that their size was fixed. While this is true, it is possible to dynamically create a new array of any size, and if necessary, abandon the original array (which, in effect, allows a programmer to dynamically alter the size of an array).

An another example, the method swap shown below will switch the values of two elements in an array of double values.

```
public static void swap (double[] list, int i, int j)
{
      double temp = list[i];
      list[i] = list[j];
      list[j] = temp;
}
```

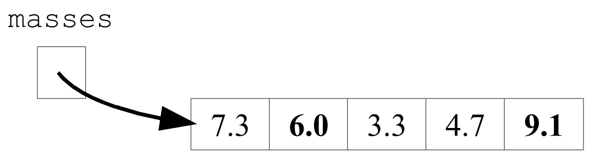
To show the effect of this code, consider the array `masses` shown below:



If we then execute the statement

```
swap(masses, 1, 4);
```

the values of `masses[1]` and `masses[4]` will be switched.



Exercises

1. What would be printed by the following program fragment?

```
int[] list = new int[4];
for (int i = 0; i < list.length; i++)
   list[i] = 3 - i;
System.out.println(list[1]+2);
System.out.println(list[1+2]);
System.out.println(list[1]+list[2]);
```

2. Suppose that an array sample has been declared as follows:

```
   int[] sample = new int[SIZE];
```

where `SIZE` is some constant value. Write one or more statements to perform each task.

a) Initialize all of the elements to one (1).
b) Switch the values at either end of the array.
c) Change any negative values to positive values.
d) Set the variable `sampleSum` to the sum of the values of all the elements.
e) Print the contents of the odd-numbered locations.

3. Write a method `max` that has one double array parameter. The method should return the value of the largest element in the array.

4. Complete the definition of the method `equals` so that it returns true if and only if its two array parameters are identical in every way.

```
public static boolean equals (double[] a, double[] b)
```

5. Write a program that repeatedly prompts the user to supply scores (out of 10) on a test. The program should continue to ask the user for marks until a negative value is supplied. Any values greater than ten should be ignored. Once the program has read all the scores, it should produce a table with the following headings (and automatically fill in the rest of the table):

```
      Score            # of Occurrences
```

The program should then calculate the mean score, rounded to one decimal place.