Lecture 3: Methods and Arrays

Sam McCauley
Data Structures, Spring 2026

Arrays

Arrays are the first data structure we will see in this course.

In Java, arrays are a fixed-length data structure to store a sequence of items of the same type. First, let's look at how to create an array.

```
1  int[] arr = new int[10]; //declares and instantiates array of 10 ints
2  double[] newArr;         //declares an array of doubles
3  newArr = new double[7]; //instantiates an array of 7 doubles
```

Note that when creating an array, you need to use the type both when instantiating and declaring—on line 1 in the above code, `int` appears on both the left and right hand side. Furthermore, you must specify the number of elements *up front*. The size of an array, and the type of elements it contains, both cannot be changed: if you want a different size or different type, you need to create a new array.

Once you create the array, you can access or modify any element of the array using the index in square brackets. Arrays are zero-indexed; the first element in the array has index 0. You can access or change these elements using the same notation you would use with a variable.

The following code prints: `10 8 7`.

```
int[] arr = new int[3]; //declares and instantiates array of 3 ints
arr[0] = 10;
arr[2] = 7;
arr[1] = arr[2] + 1;
System.out.print(arr[0]);
System.out.print(" ");
System.out.print(arr[1]);
System.out.print(" ");
System.out.println(arr[2]);
```

You can use the following notation to create the array and set its values at the same time. This is occasionally useful for small arrays that contain constants.

```
int[] arr = {2,4,6};
System.out.println(arr[1]);  //prints 4
```

If you access an index that is larger than the size of the array, or less than 0, you will get an error.

```java
int[] arr = new int[3]; //declares and instantiates array of 10 ints
arr[3] = 0; //Error!  The available indices are: 0, 1, 2
```

The size of the array can be accessed using `.length`.

```java
int[] arr = new int[3]; //declares and instantiates array of 3 ints
System.out.println("The size of the array is");
System.out.println(arr.length);
```

Note that Java does *not* automatically print an array for you.[1] The following code will print, essentially, gibberish.

```java
int[] arr = {2,4,6};
System.out.println(arr);  //on my machine, prints: [I@7ad041f3
```

Exercise: Print Contents of an Array. Let's write a method that prints the contents of an integer array, since Java won't do it for us. We'll call the method `printArray`.

This method doesn't return anything (it just prints the array). So we'll set the return type to `void`.

The only parameter we need is the array itself; we'll call this `theArray`. Putting these parts together, we have the following method declaration:

```java
public static void printArray(int[] theArray) {
}
```

We'll use a while loop to print the contents of the array. When we're looping, we need to keep track of the current index in a variable. We'll call this variable `index`.

Since arrays are 0-indexed, we'll start by setting `index` to 0. Each time we loop, we'll print the current entry of the array, then print a space, and then increment the index. The loop will continue iterating as long as index is less than the length of the array. After the loop, we'll print a new line.

Let's put this plan into the code.

```java
public static void printArray(int[] theArray) {
    int index = 0;
    while(index < theArray.length) {
        System.out.print(theArray[index]);
        System.out.print(" ");
        index += 1;
    }
    System.out.println();
}
```

[1]There is a standard way to print the contents of an array, but it requires a library call. For now, we'll print arrays manually.

Strings

Let's discuss how strings work in Java, along with a few quality-of-life shortcuts.

`String`s can be declared and instantiated just like any other variable. `String` constants in Java are put in quotes. Note that `String` is capitalized.

```java
    String message = "Hello World!";
    System.out.println(message);
```

Strings can be concatenated using the + operator.

```java
    String message1 = "Hello ";
    String message2 = "World!";
    System.out.println(message1 + message2); //prints: Hello World!
```

If you take the sum of a `String` and a non-string (let's say: an integer variable), the latter will first be converted into a `String`; then the `String`s will be concatenated. This is often useful for printing variables in Java.

```java
int x = 100;
//prints: The value of x is: 100
System.out.println("The value of x is: " + x);
```

String length. You can access the length of a string using `.length()` (note that unlike arrays, we are required to use parentheses here—that's because this is a method call).

For example:

```java
    String message = "Hello World!";
    System.out.println(message.length()); //prints 12
```

Special Characters. There are several special characters for `String` constants. Perhaps the most useful is the newline character `\n`. The `\n` is a single character, representing a new line. Putting new lines in your strings can be very useful for helping to format print statements.

Consider the following example:

```java
    System.out.println("Line 1");
    System.out.print("Line 2\nLine 3\n");
```

This prints:

```
Line 1
Line 2
Line 3
```

There are in fact quite a few special characters, each of which starts with the “escape character” `\`. Some other useful ones are `\t` which is a “tab” character (good for printing tables), `\'` and `\"` which allow you to make strings that contain apostrophes and quotation marks, and `\\` which allows you to include a backslash in a string.

Wrapping up Java Operations

Other Java Operations. There are several shortcuts for basic arithmetic on variables in Java.

The notation `a += b` is a shortcut for `a = a + b`; other operations like `-=`, `*=`, `/=` are defined similarly. `a++` is a shortcut for `a = a + 1`. Lines 2–4 of the following code all do the same thing.

```java
1 int x = 1;
2 x = x + 1; //x is now 2
3 x += 1;    //x is now 3
4 x++;       //x is now 4
```

You can also use − to get the negative of an integer.

There are some further operators that deal with the binary representation of the variable—manipulating the actual bits that constitute the number. Here I am referring to the operators: &, |, ~, <<, and >>. You do not need to know these now, but we may see them later in the course.

Logical Operators and Short-Circuit Evaluation. In your past experience programming, you have probably seen ways to combine multiple conditionals in, say, an `if` statement. For example, maybe we have code we want to execute if either `x < 0` or `x >= 100`.

In Java, there are three operators that are useful in these contexts.

- && (and): true if *both* conditional expressions are true
- `||` (or): true if *at least one* of the conditional expressions is true. (Could be both, or could be just one.) The key to press to make this character is located above the enter key.
- `!` (not): converts true into false and false into true

Let’s look at some examples.

```java
if(x >= 10 && x <= 99) {
    System.out.println("x has two digits");
}
```

```java
if( !(x >= 10 && x <= 99) ) {
    System.out.println("x does not have two digits");
}
```

```java
//c has type char
if(char == 'a' || char == 'e' || char == 'i'
|| char == 'o' || char == 'u') {   //this line break is legal Java
    System.out.println("Char is a vowel");
}
```

Let's say that a number is "interesting" if it is positive, and it is either a prime number or a perfect square. Let's say we have already created methods `isPrime()` and `isSquare()` that take in an integer parameter, and return a `boolean` determining if the number is prime or square. Then the following code determines if a number is interesting.

```java
if(x > 0 && (isPrime(x) || isSquare(x)) ) {
    System.out.println("Look at this interesting number: " + x);
}
```

One important hint relating to the above example: ***always use parentheses if you are using multiple logical operators.*** How precedence works with these operators can be subtle and counterintuitive—parentheses ensure that your code is doing what you expect.

Short-Circuit Evaluation. Java takes a shortcut internally which occasionally has implications on how code is run. This mostly comes up when your conditionals have *side effects*: for example, you call a method to test a variable, and that method also prints to the screen (see the final example below).

If two expressions are separated by &&, and the first is false, we already know that the final expression is false—we don't need to look at the second one. Java takes this shortcut, and does not evaluate the rest of the expression.

If two expressions are separated by `||`, and the first is true, we already know that the final expression is true—we don't need to look at the second one. Java takes this shortcut, and does not evaluate the rest of the expression.

Let's look at the following example.

```java
public static boolean isEven(int x) {
    System.out.println("Testing if " + x + " is even.");
    return (x % 2 == 0);
}
public static boolean isLarge(int x) {
    System.out.println("Testing if " + x + " is large.");
    return x > 10;
}
```

```
 9  public static void main(String[] args) {
10      int num = 21;
11      if(isEven(num) && isLarge(num)) {
12          System.out.println(num + " is even and large");
13      }
14      int num2 = 6;
15      if(isEven(num2) || isLarge(num2)) {
16          System.out.println(num2 + " is either even or large, or both");
17      }
18  }
```

Let's trace through the execution of this code. Execution starts at the beginning of the `main` function, on line 10. Java creates num and stores 21 in num.

On line 11, Java calls `isEven(num)` to determine if num is even. This prints "Testing if 21 is even" to the screen. This method evaluates to false. Java looks at the && and determines that the whole `if` statement is false: it does not call `isLarge(num)`.

The program proceeds to line 14; Java creates num2 and stores 6.

On line 15, Java calls `isEven(num2)` to determine if num2 is even. This prints "Testing if 6 is even" to the screen. This method evaluates to true. Java looks at the `||` and determines that the whole `if` statement is true: it does not call `isLarge(num2)`. The program proceeds to line 16, and prints to the screen "6 is either even or large, or both."

Overall, the output of the program is:

```
Testing if 21 is even
Testing if 6 is even
6 is either even or large, or both.
```

Java Types

We've already seen that each variable must be assigned a *type* in Java, but we have not said much about how these types interact.

Mixing Types. What happens when an expression has multiple types?

```
    int x = 4;
    double y = 2.1;
    System.out.println(x + y);
```

We actually already saw one example of this: if one operand is of type `String`, the other is converted to a `String` and they are then concatenated.

The above example is similar. If a `double` and `int` are summed (or subtracted, multiplied, divided, etc.), the result is a `double`.

Mixing `doubles` with `ints`, or arbitrary values with `Strings`, is the most common way you'll see types mixed. The Java documentation contains the full list of rules for how this works in general: https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html#jls-5.1.2

Similar rules work for assignment. The following code runs in Java.

```java
    int x = 4;
    double y = x + 1;
    System.out.println(y); //prints 5.0
```

However, note that Java will give an error if you are going to *lose* information due to type change.

```java
    double x = 4.2;
    int y = x - 1; //the .2 is dropped; the compiler gives an error
```

To make this work, you need to explicitly change the type of the data (in this case, turn `x` into an integer). See the discussion below on "Casting."

Types of Constants. An important sticking point is that constants have types in Java as well. Usually this is fairly clear: for example, `"Hello"` is a `String`. Do bear in mind that single quotes, such as: `'A'`, represent a `char` rather than a `String`.

Numerical values with no decimal are assumed to be `ints`. This leads to the following situation:

```java
int feet = 10;
double yards = feet/3;
System.out.println(yards);  //prints 3.0
```

What happened here? Let's look at what happens in line 2. Java divides `feet` by 3. That's dividing an `int` by an `int`, so Java uses *integer division*, and obtains 3. Then, Java happily stores this integer value in `yards`.

To avoid this issue, we need to make sure that `feet/3` is not integer division. We could change the type of `feet` to `double` and we'd be all set. Alternatively, we can instead divide by 3.0.

```java
int feet = 10;
double yards = feet/3.0;
System.out.println(yards);  //prints 3.33333...
```

Here, when Java sees `feet/3.0`, it sees an operation between a float and an integer. Java first turns `feet` into a `double`, then does the division for `doubles` (retaining the fractional part), storing the result in `yards`.

Below, we will see that we can use casting to solve this problem as well.

Casting. Occasionally we will want to change the type of data. Note that a *variable* cannot have its type changed: if we instantiate `double x`, then x will stay a `double` for as long as it is around. However, it may be useful to change the type of the contents—the information itself. This is called *casting*. We say something like: "cast it to an `int`" to mean "take the value, and convert it to an integer."

The notation to change the type of data is to put the new type in parentheses immediately before the data. Consider the following example (fixing the error from above)

```
    double x = 4.2;
    int y = (int)x - 1;
    System.out.println(y);
    System.out.println(x);
```

When Java sees `(int)x`, it takes `4.2` and converts it into an `int`, with value 4. It subtracts 1 to obtain 3, and stores 3 in y (this was integer subtraction, since both sides are `int`s). Note that the value of x is unchanged: Java looked at x to get the value 4.2, and it was this value that was cast to an integer.

Casting is perhaps the clearest way to explicitly tell Java to avoid integer division as well—compare this code with the example above).

```
int feet = 10;
double yards = (double)feet/3;
System.out.println(yards);  //prints 3.33333...
```