ACTIVITY **3.4.1**

Investigating Sort

INTRODUCTION

Sorting algorithms are part of many computer science applications. Shopping apps sort their products by cost, YouTube sorts results by relevance, and games sort users' high scores. To appreciate the power of modern-day sorting algorithms and computation in general, let's start with a short walk down memory lane to when languages were not as versatile as they are today.

Materials

- Computer with Android™ Studio
- Android™ tablet and a USB cable, or device emulator

Procedure

Part I: Sequential Programming

1. The program below is written in the C programming language. Java is an extension of C, so you should be able to read the algorithm below. The `printf` command prints the given value to the screen:

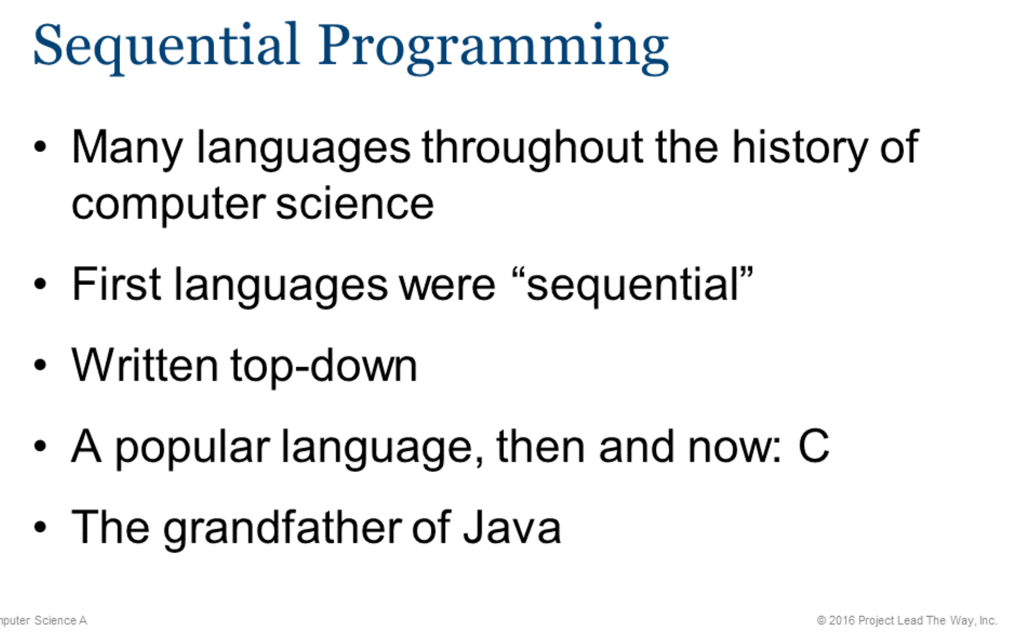
```
int numbers[] = {4,8,17,2,9,13,6,21,7,1};
int i;
int j;
int temp;

for (i = 1 ; i <= sizeof(numbers) - 1; i++)
{
  j = i;
  while (j > 0 && numbers[j] < numbers [j-1])
  {
    temp = numbers[j];
    numbers[j] = numbers[j-1];
    numbers[j-1] = temp;

    j--;
 }
}

for (i = 0; i< 10; i++)
{
  printf("%d\n", numbers[i]);
}
```

As best you can, describe what the code above does. Focus on what the nested loops do. Review the slideshows.



Sequential Programming

- Many languages throughout the history of computer science
- First languages were “sequential”
- Written top-down
- A popular language, then and now: C
- The grandfather of Java

Computer Science A

Accompanies Step 1 of Activity 3.4.1 Investigating Sort

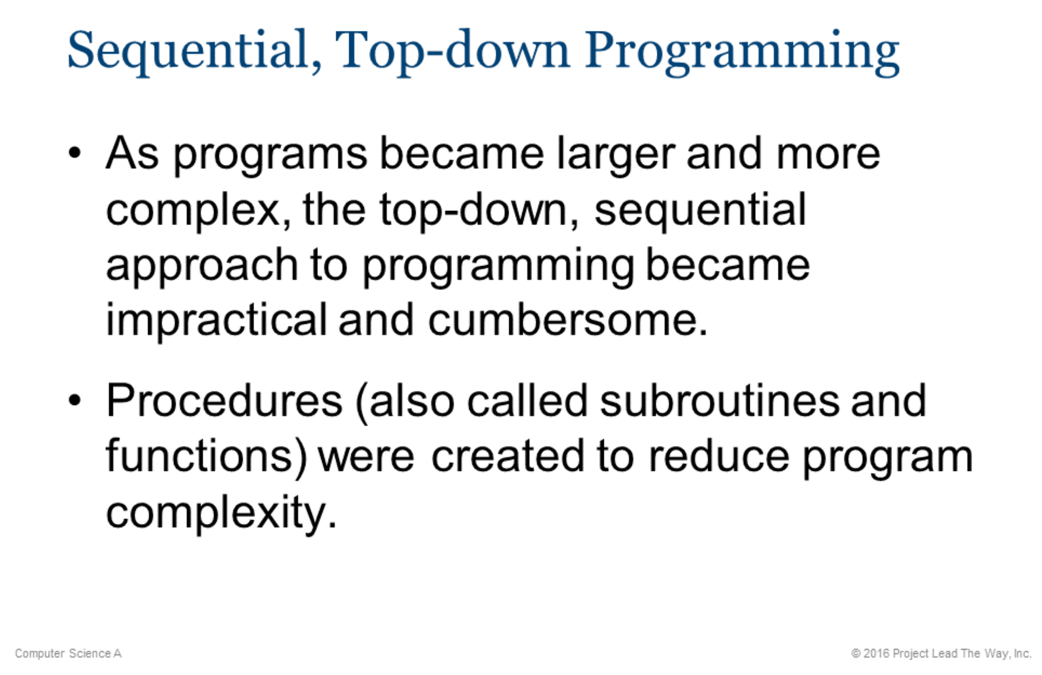
In the evolution of Computer Science, many languages have been invented.

- An early version of COBOL in the 1940s; used primarily for business applications
- Fortran in the 50s; used for scientific applications
- C in the 70s; originally used to write a new version of the UNIX operating system, became a general purpose language (Java is built on C++ which is built on C)

The list of programming languages throughout history is long. If you are curious, visit https://en.wikipedia.org/wiki/History_of_programming_languages

Early programming languages were “sequential” languages; execution began at the first line of code and continued one line after the other to the last line. There are no objects, no events that interrupt program flow, no parallel processing, and with the first programming languages, no subroutines. Programmers developed applications without much thought to code re-use, generalization or even maintenance. Furthermore, data representations were custom designed and built for the single purpose the programmer had in mind. When new requirements came up, often entire software applications had to be rebuilt nearly from scratch.

Even with this “limited” way of programming, the language C is still very much in use. A main benefit is its scalability: programmers can write very small, fast, light-weight applications and also large, complex computational systems.

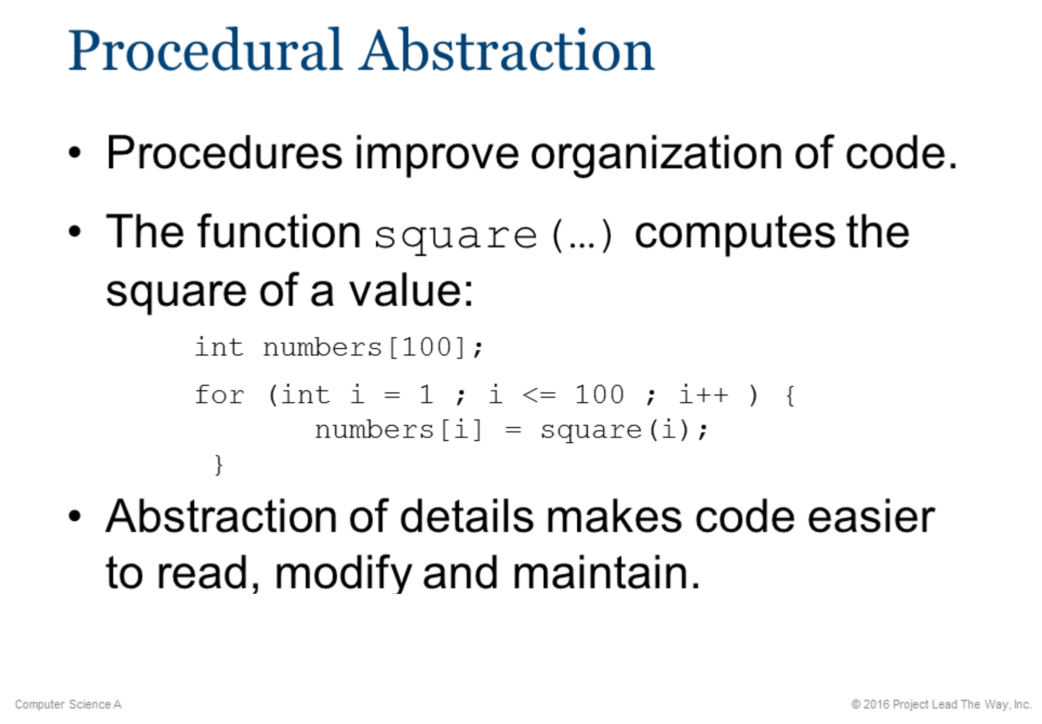


Sequential, Top-down Programming

- As programs became larger and more complex, the top-down, sequential approach to programming became impractical and cumbersome.
- Procedures (also called subroutines and functions) were created to reduce program complexity.

Computer Science A

© 2016 Project Lead The Way, Inc.



Procedural Abstraction

- Procedures improve organization of code.
- The function `square(…)` computes the square of a value:

```
int numbers[100];
for (int i = 1 ; i <= 100 ; i++ ) {
        numbers[i] = square(i);
 }
```

- Abstraction of details makes code easier to read, modify and maintain.

Computer Science A

© 2016 Project Lead The Way, Inc.

The square function is a simple example of procedural abstraction, the idea that code can be collected in a way to make a program easier to read and maintain, removing detail from higher-level parts of the program.

The notion of procedural abstraction gave rise to Object-Oriented Programming, which is founded on the idea that not only can functions be collected to help organize a program, but the data itself can be collected and organized.

In Java, the methods you have used and written are all examples of procedural abstraction.

2 C is an example of a **sequential programming** language written in a “top-down” manner. Looking at the code in step 1, if a programmer needed to sort letters into alphabetical order instead of integers, how would all of the code need to be changed?

3 To hide some of the details in the program using **procedural abstraction**, what lines in the program might be collected and organized into a new function? Be sure to give the function an appropriate name.

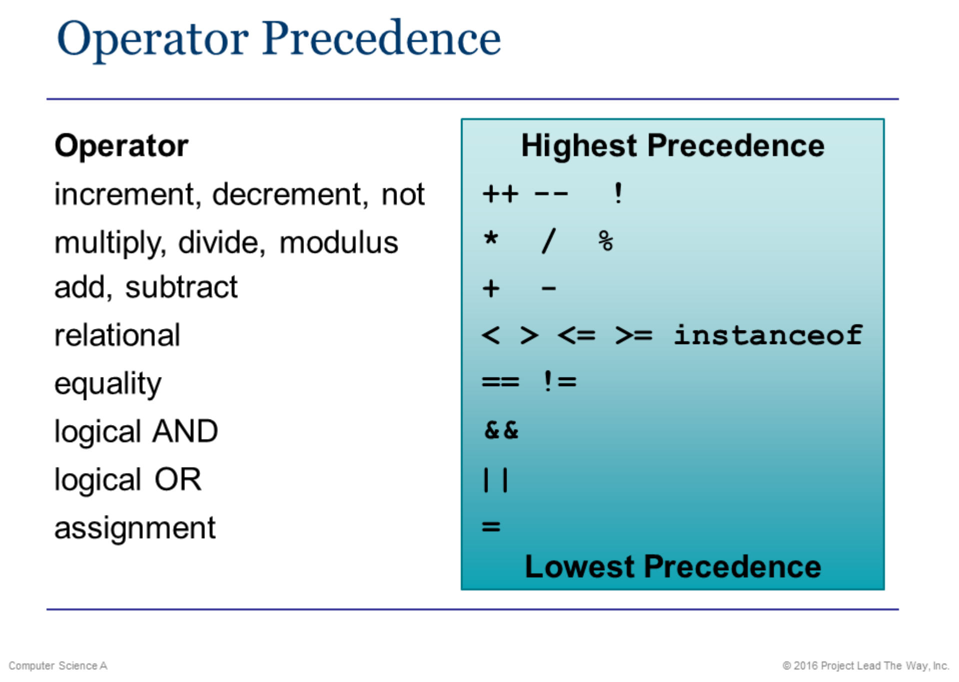
4 To create an object-oriented version of this program, what objects and methods might you create? What methods could be overloaded?

sequential programming
The style of writing code in a “top-down” format; statement execution occurs in order from the beginning to the end of the code.

procedural abstraction
The paradigm of hiding multiple or complex steps in a procedure to make code readable and reusable.

Part II: Operator Precedence

5 Using the code in Part 1, ignore the `printf` statement and identify all of the operators you can find. List them here.



Operator Precedence

Operator	Highest Precedence
increment, decrement, not	`++ -- !`
multiply, divide, modulus	`* / %`
add, subtract	`+ -`
relational	`< > <= >= instanceof`
equality	`== !=`
logical AND	`&&`
logical OR	`||`
assignment	`=`
	Lowest Precedence

Computer Science A

© 2016 Project Lead The Way, Inc.

6 Reorder the operators from the previous step in order of precedence, grouping those with the same precedence.

7 In the following statement, identify the operators in terms of highest precedence to lowest precedence.

```
if (a + b > c * d && e < f || g != h % (int)i && !j)
```

8 Using the default precedence, rewrite the statement, adding parentheses to make the statement easier to read.

9 How are operators evaluated in a statement when the operators have equal precedence? Use the following statement as an example:

```
a / b % c * b
```

Part III: Sort Analysis

10 Obtain and import the app *3.4.1WhiteboardAppSort* as directed by your teacher.

11 In the `onCreate` method in *WhiteboardAppActivity*`.java`, create an array list of strings called `myList`.

12 Add a few strings to the list. They can have any value, except they should not be sorted in any particular order. The strings can be something like names of people, favorite foods, or sports teams.

13 Open *SelectionSort.java* and determine how the static `sort` method is used. Use this class and method to sort your list in the `onCreate(…)` method.

14 Display the sorted list in your app using either a `for` loop or an enhanced `for` loop.

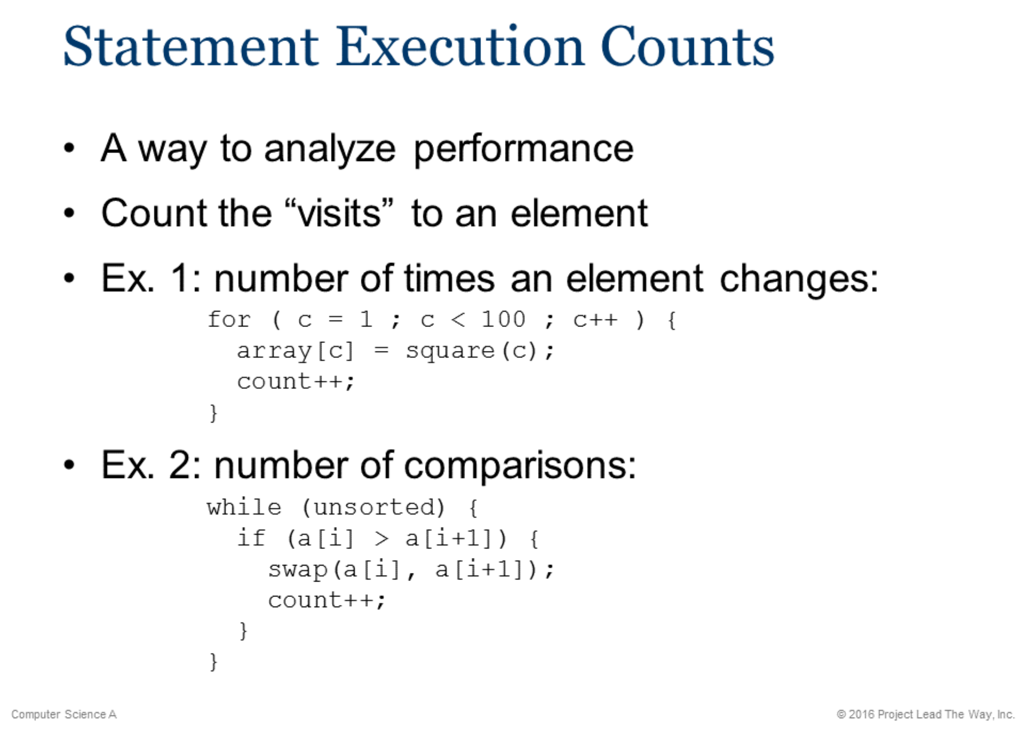
15 To confirm that your list is sorting properly, run your app a few times, changing some string values.

16 Now that selection sort is working, you will do some run-time analysis using large lists of strings. Creating a large list of anything in a program can be tedious and time consuming, but not if you use the following algorithm in place of your `myList.add(…)` lines:

```
1: String letters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
2: int wordLength = 5;
3: int numWords = 10;
4:
5: for (int i = 0; i < numWords; i++) {
6:     int r = (int)( Math.random() * (letters.length()
       - wordLength - 1));
7:     myList.add(letters.substring(r, r+wordLength));
8: }
```

17 Test your app to make sure you have 10 strings displayed and sorted. Explain what the algorithm does at lines 5–8, providing detail about how the value of `r` is determined.

18 (Slide 6) Open *SelectionSort.java* and add a **statement execution count** to the `sort(…)` method wherever a comparison is made.



Statement Execution Counts

- A way to analyze performance
- Count the "visits" to an element
- Ex. 1: number of times an element changes:

```
for ( c = 1 ; c < 100 ; c++ ) {
  array[c] = square(c);
  count++;
}
```

- Ex. 2: number of comparisons:

```
while (unsorted) {
  if (a[i] > a[i+1]) {
    swap(a[i], a[i+1]);
    count++;
  }
}
```

Computer Science A

© 2016 Project Lead The Way, Inc.

19 Modify the `sort(…)` method again to return the statement execution count to the calling method and adjust the calling method appropriately.

20 Change the data that is displayed in your app to be *only* the execution count of the selection sort.

21 Use nested looping to generate different sets of words and then sort them. Use the outer loop to generate the numbers of words shown in the table below and use the inner loop to run the selection sort. Collect the following performance data (the app may take a while to perform all of the sorts):

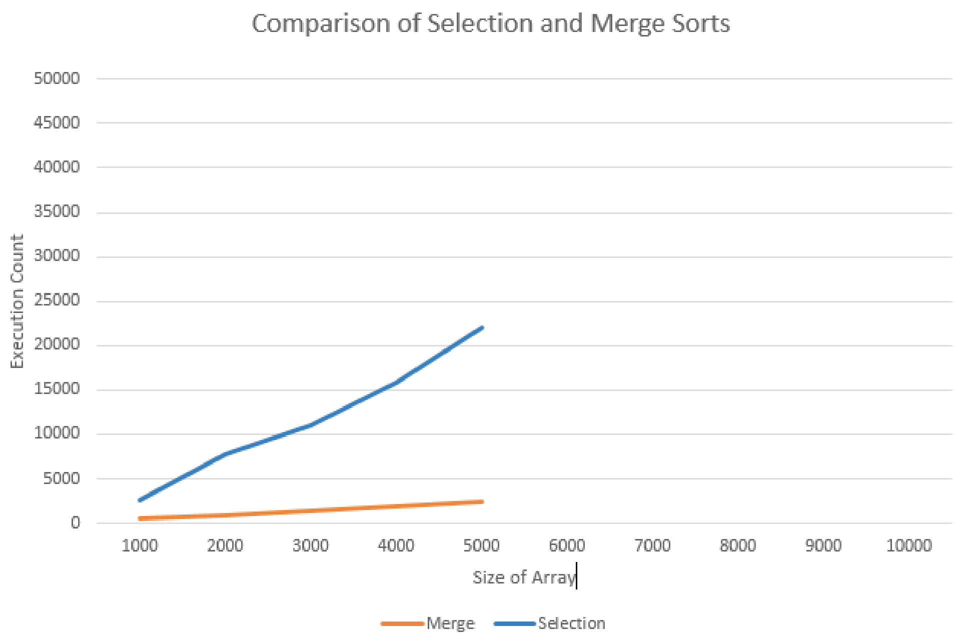
Number of Words	Selection Sort Statement Execution Count
1000	
2000	
3000	
4000	
5000	

22 Open *MergeSort.java* and add a statement execution count to the algorithm wherever a comparison is made.

23 Change the calling method to use a merge sort instead of the selection sort and collect the following data:

Number of Words	**Merge Sort Statement Execution Count**
1000	
2000	
3000	
4000	
5000	

24 Although exact data will vary from test run to test run, the following pattern emerges when graphing the execution counts of selection and merge sorts:



25 Even though you have no data beyond 5000 strings, extrapolate (extend and estimate) to predict an approximate count when sorting 10,000 strings for both the selection and merge sorts.

26 (Challenge) Observe the three online sort algorithms— **insertion, selection, and merge sorts**—and analyze them in terms of search times.

a. Use an online stopwatch to record approximate sort times (search for “stopwatch” in Google). You can control the speed of the animation on the Comparison Sort page with the Animation Speed slider.

b. Record the results of a few performance runs.

   What performance conclusions can you make about the three methods of sorting?

   Which sort methods are similar in terms of performance and what makes them similar?

Part IV: Other Statement Execution Counts

27 Using each of the following algorithms, predict the execution count:

a.
```
for(int i = 0; i < 10; i++) {
    for (int j= 0; j < 5; j++) {
        count++;
    }
}
```

- Statement execution count:
- Describe how you predicted this.

b.
```
for(int i = 0; i < 50; i = i+2) {
   for (int j= 0; j < 10; j = j+2) {
        count++;
   }
}
```

- Statement execution count:
- Describe how you predicted this.

c.
```
for(int i = 1; i <= 10; i++) {
   for (int j = 1; j <= i; j++) {
        count++;
   }
}
```

- Statement execution count:
- Describe how you predicted this.

CONCLUSION

1. How does a sequential programming approach to solving a problem limit a solution's reuse?
2. How is the selection sort similar to sequential programming?
3. Write out the state of the array for all passes of the *outer* loop in the sorting algorithm from step 1.
4. (Challenge) Generalize your prediction algorithm from Step 27c.
   a. Predict the execution count for the following:

```
for(int i = 1; i <= n; i++) {
    for (int j= 1; j <= i; j++) {
         count++;
    }
}
```

   b. Determine whether it is the same execution count for the nested loops below.

```
for(int i = 0; i < n; i++) {
    for (int j= 0; j < i; j++) {
         count++;
    }
}
```