PTI CS 103

Prison Teaching Initiative

Mar 26, 2019

Contents

1

Introduction

Todo:

- Put together all references at the end
- Ask Sedgewick for permission to copy HelloWorld
- Put in references to lecture names for hardware and software instead of lecture 3 and lecture 4 (respectively)
- Add pictures
- Add course expectations (e.g. grading)
- Add more discussion questions
- Add section for the algorithmic thinking activity

1.1 Overview of course

Knowing just a little bit of computer science can get you started right away in actual applications. One of the goals of this course is to learn about the fascinating subject of computer science. Another is to develop algorithmic thinking skills that will help with day-to-day critical problem-solving skills. But perhaps the most important goal of the course is to develop coding skills, which will not only open up new job opportunities but also make you more effective in most areas of business.

In the first semester, we will spend the first two classes of each week on computer science theory and special topics. The final day of each week will be a lab day, where we actually start practicing coding skills.

In the second semester, we will start focusing more on practical coding, with a single day a week for theory and 2 lab periods per week for coding.

Broadly, we will cover the following topics:

- **How modern computers work**
  - Hardware
  - Software
  - Computer networks and information systems
- **Algorithms for quickly solving complex problems**

  - Searching
  - Sorting
- **Data structures**
  - Arrays
  - ArrayLists
- **Applications of Computer science**
  - Basic coding in Java
  - How to use productivity software

1.2 Brief history of computer science

Timeline:

- Invention of the abacus (2700-2300 BC, Sumerians)
- Design of first modern-style computer (Charles Babbage, 1837)
- Design of first computer algorithm (Ada Lovelace, 1843)
- Invention of first electronic digital computer (Konrad Zuse, 1941)
- Invention of the transistor (Bell labs, 1947)
- Invention of the first computer network (early Internet) (DARPA, 1968)
- Invention of the World Wide Web (Sir Tim Berners-Lee, 1990)

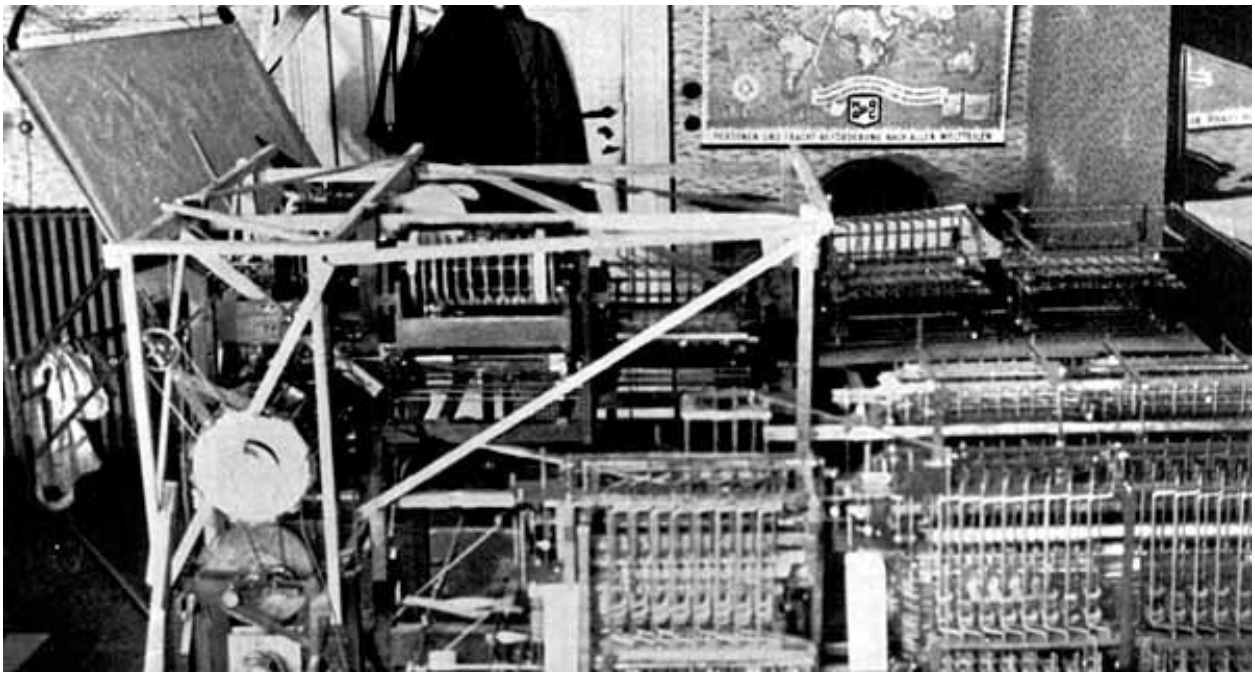

Fig. 1: Construction of Konrad Zuse's Z1, the first modern computer, in his parents' apartment.

1.3 Components of a computer

A computer is an electronic device used to process data. Its basic role is to convert data into information that is useful to people.

There are 4 primary components of a computer:

- Hardware
- Software
- Data
- User

1.3.1 Hardware

Computer hardware consists of physical, electronic devices. These are the parts you actually can see and touch. Some examples include

- Central processing unit (CPU)
- Monitor
- CD drive
- Keyboard
- Computer data storage
- Graphic card
- Sound card
- Speakers
- Motherboard

We will discuss these components in more detail in lecture 3.

1.3.2 Software

Software (otherwise known as "programs" or "applications") are organized sets of instructions for controlling the computer.

There are two main classes of software:

- Applications software: programs allowing the human to interact directly with the computer
- Systems software: programs the computer uses to control itself

Some more familiar applications software include

- Microsoft Word: allows the user to edit text files
- Internet Explorer: connects the user to the world wide web
- iTunes: organizes and plays music files

While applications software allows the user to interact with the computer, systems software keeps the computer running. The operating system (OS) is the most common example of systems software, and it schedules tasks and manages storage of data.

We will dive deeper into the details of both applications and systems software in lecture 4.

Fig. 2: Examples of hardware components of a personal computer.

1.3.3 Data

Data is fundamentally information of any kind. One key benefit of computers is their ability to reliably store massive quantities of data for a long time. Another is the speed with which they can do calculations on data once they recieve instructions from a human user.

While humans can understand data with a wide variety of perceptions (taste, smell, hearing, touch, sight), computers read and write everything internally as “bits”, or 0s and 1s.

Computers have software and hardware which allow them to convert their internal 0s and 1s into text, numerals, and images displayed on the monitor; and sounds which can be played through the speaker.

Similarly, humans have hardware and software used for converting human signals into computer-readable signals: a microphone converts sound, a camera converts pictures, and a text editor converts character symbols.

1.3.4 Users

Of course, there would be no data and no meaningful calculations without the human user. Computers are ultimately tools for making humans more powerful.

As we will see in the next section, however, different types of computers have different roles for the user.

1.4 Types of computers

1.4.1 Supercomputers

These are the most powerful computers out there. The are used for problems that take a long time to calculate. They are rare because of their size and expense, and therefore primarily used by big organizations like universities, corporations, and the government.

The user of a supercomputer typically gives the computer a list of instructions, and allows the supercomputer to run on its own over the course of hours or days to complete its task.

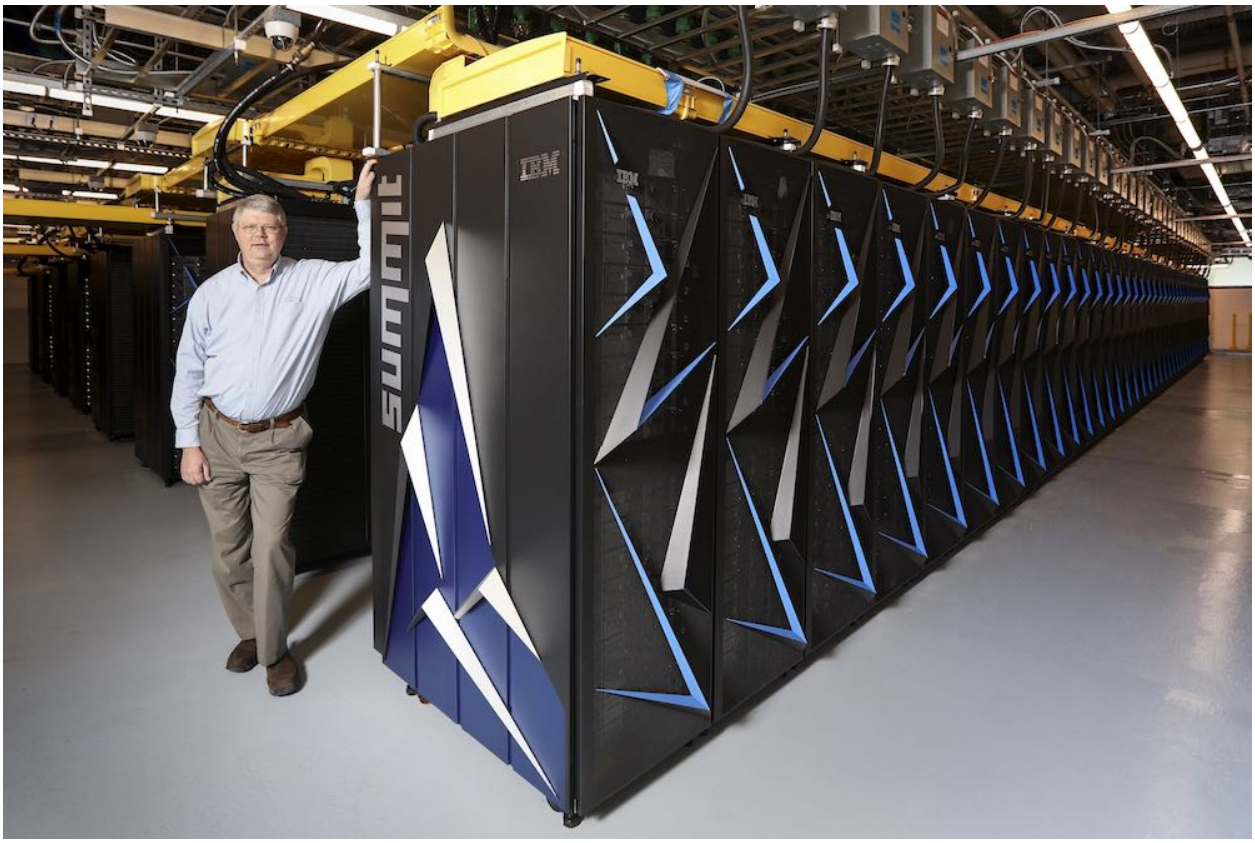

Fig. 3: Summit, a world-class supercomputing cluster at Oak Ridge National Laboratory in Tennessee.

1.4.2 Mainframe computers

Although not as powerful as supercomputers, mainframe computers can handle more data and run much faster than a typical personal computer. Often, they are given instructions only periodically by computer programmers, and then run on their own for months at a time to store and process incoming data. For example, census number-crunching, consumer statistics, and transactions processing all use mainframe computers.

1.4.3 Personal computers

These are the familiar computers we use to interact with applications every day. Full-size desktop computers and laptop computers are examples.

1.4.4 Embedded computers

In the modern "digital" age, nearly all devices we use have computers embedded within them. From cars to washing machines to watches to heating systems, most everyday appliances have a computer within them that allows them to function.

1.4.5 Mobile computers

In the past 2 decades, mobile devices have exploded onto the scene, and smartphones have essentially become as capable as standalone personal computers for many tasks.

1.5 Why computers are useful

Computers help us in most tasks in the modern age. We can use them, for example, to

- write a letter
- do our taxes
- play video games
- watch videos
- surf the internet
- keep in touch with friends
- date
- order food
- control robots and self-driving cars

  Question: What are some other tasks a computer can accomplish?

This is why the job market for computer scientists continues to expand, and why computer skills are more and more necessary even in non-computational jobs.

According to a Stackoverflow survey from 2018, 9% of professional coders on the online developer community Stack Overflow have only been coding for 0-2 years. This demonstrates two things:

1. The job market for people with coding skills is continually expanding
2. It doesn't take much to become a coder

Some examples of careers in computer science include

- IT management / consulting
- Game developer
- Web developer
- UI/UX designer
- Data analyst
- Database manager

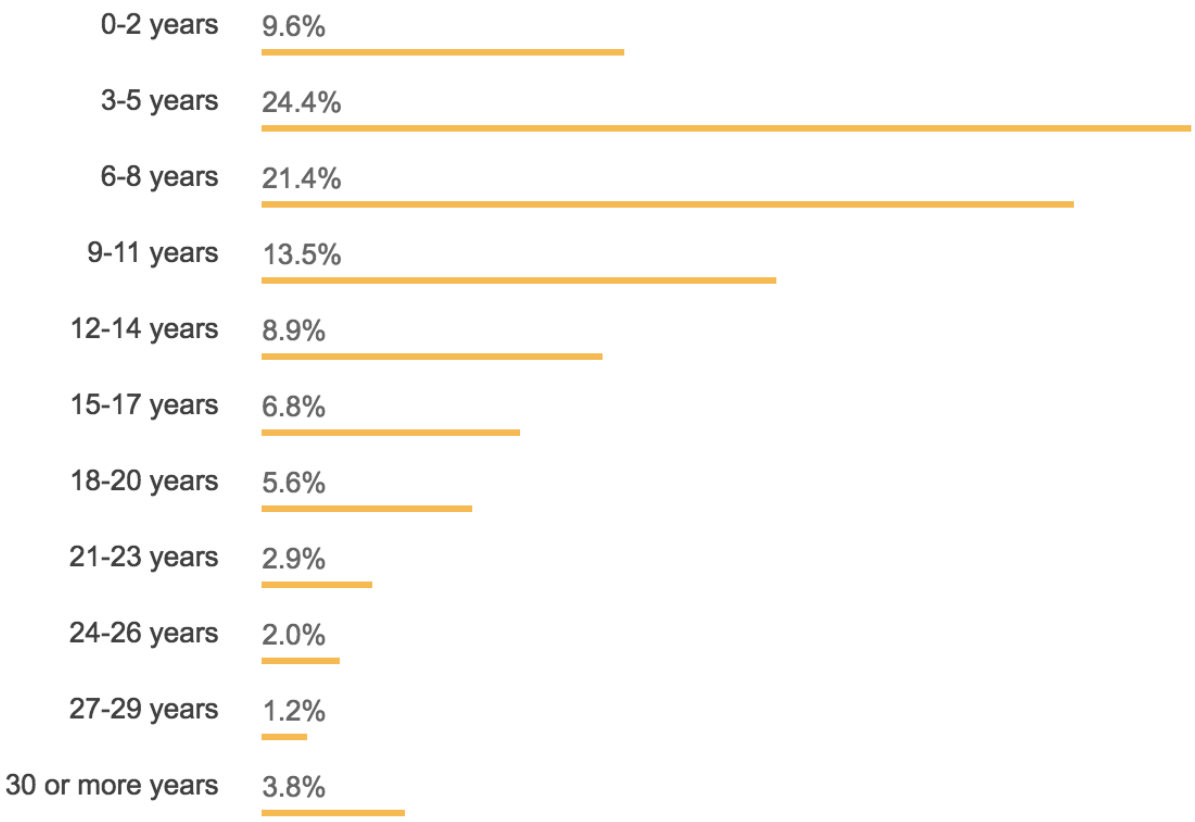

Fig. 4: Percentage of Professional Developers on Stack Overflow with various levels of coding experience.

1.6 First program

Please refer to the attached handout from the online resource corresponding to‘Computer Science: An Interdisciplinary Approach <https://introcs.cs.princeton.edu/java/11hello/>‘_, Robert Sedgewick and Kevin Wayne, “Your First Jaba Program: Hello World”.

> **Exercise:** Create your own program, `HelloMe.java`, that prints out “Hello `name`” with your own name in place of `name`.

1.7 References

- Computer Science: An Interdisciplinary Approach, Robert Sedgewick and Kevin Wayne.
- University of Wisconsin-Madison CS 202 Lectures, Andrea Arpaci-Dusseau.

2

Numbering Systems

Topics

- *Overview*
- *Decimal Numbers*
- *Radix Decomposition*
- *Binary Numbers*
- *Binary Arithmetic*
- *Binary to Decimal Conversion*
- *Decimal to Binary Conversion*
- *Conclusion*

2.1 Overview

How do we represent numbers? Based on how we represent numbers, certain operations can be easier or harder. Today, we'll consider two representations: decimal and binary numbers. Decimal numbers (decimal notation or base 10) are what we're all used to seeing in our daily lives. For example we could consider the string of numbers:

$$19.874$$

And we know this represents the value nineteen thousand eight hundred seventy four. However, we could consider representing numbers differently. Depending on how we represent numbers specific operations become easier or harder. Let's go back to our example and now we'll multiply by 10.

$$\begin{array}{r} 19,874 \\ \times \quad 10 \\ \hline 198,740 \end{array}$$

2.2 Decimal Numbers

When we see numbers in our daily lives there's an implicit assumption that these numbers are decimal numbers, whether this is a price, the temperature, or time. We can be explicit about what representation we are using by

subscripting our numbers with the base.

$$\mathbf{19,875_{10}} \text{ or } \mathbf{12.38_{10}}$$

In this class, unless otherwise stated we'll consider all numbers to be represented in base 10.

2.3 Radix Decomposition

How do we actually go from numbers in base 10 to finding their value? We can break down the number by each position. Let's go back to our running example:

$$\begin{aligned} 19,874 &= 1 \times 10,000 & +9 \times 1000 + 8 \times 100 & & +7 \times 10 + 4 \times 1 \\ &= 1 \times 10^4 & +9 \times 10^3 + 8 \times 10^2 & & +7 \times 10^1 + 4 \times 10^0 \end{aligned}$$

How do we know the value is nineteen thousand eight hundred seventy four? We can break it down digit by digit and add together the values. We have 4 in the 1s (the right-most position), 7 in the 10's position, 8 in the hundreds position, 9 in the thousands, and 1 in the ten-thousands position. Or in other words, at each successive position we multiply the value of the digit at that position by the next power of 10 to determine its value. This even works for non-whole numbers. Consider the number:

$$\begin{aligned} 12.38 &= 1 \times 10 & +2 \times 1 + \frac{3}{10} & & +\frac{8}{100} \\ &= 1 \times 10^1 & +2 \times 10^0 + 3 \times 10^{-1} & & +8 \times 10^{-2} \end{aligned}$$

We find 1 in the tens position, 2 in the ones position, 3 in the tenths position, and 8 in the hundredths position. For non-whole numbers we treat every digit to the right of the decimal (or radix) point exactly the same as we do for whole numbers. Everything to the left of the radix point we successively divide by the next power of ten. Now that we know how to determine the value of a decimal number, how can we do this for other number representations. We'll now transition to considering binary numbers.

2.4 Binary Numbers

What are binary numbers? They're just another way to represent numbers; however, instead of having ten digits (one to nine) we have two bits (zero and one). Last week, we learned about computer hardware. A computer's primary purpose is to compute; so we need to be able to store numbers on a computer's hardware. Without getting too technical, a computer represents numbers in a sequence of transistors, each storing an electrical charge. These charges can be on (high voltage) or off (low voltage) or somewhere inbetween. However, like with a light bulb that may burn brighter or be dimmer based on the incoming charge, we only consider if the state is on or off.

In effect, computers represent numbers in binary. To better understand how computers work, we'll practice with binary numbers. Let's consider a few example numbers.

$$\mathbf{1_2 = 1_{10}} \text{ and } \mathbf{11_2 = 3_{10}} \text{ and } \mathbf{101_2 = 5_{10}} \text{ and } \mathbf{110.01_2 = 6.25_{10}}$$

Here we see how we represent 1, 3, 5, and 6.25 in binary notation.

2.5 Binary Arithmetic

Just as we perform arithmetic on decimal numbers, we can perform arithmetic on binary numbers. The process is exactly the same, except we work with bits instead of digits. Let's start by looking at addition and multiplication.

$$\begin{array}{r} 10110101 \\ +\ 1010110 \\ \hline 100001011 \end{array}$$

$$\begin{array}{r} 110101 \\ \times\ \ 1101 \\ \hline 110101 \\ 11010100 \\ 110101000 \\ \hline 1010110001 \end{array}$$

Here we see how to add the binary representations of 181 and 86 and get 267 as the result. We also see how to do long form multiplication of 53 and 13 to get 689. You'll notice this is exactly the same as for decimal numbers but we force ourselves to only work with bits, carrying the one to the next place when necessary. Both subtraction and division work similarly.

Now we'll turn our attention to three important arithmetic operations on binary numbers **and**, **or**, and **exclusive or**. These are generally called bit-wise operations as they apply to each bit place without considering the result of other placements. For these operations, we'll pad the shorter of the two numbers with zeros when needed.

A	B	A and B	A or B	A xor B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Let's practice a few examples!

$$\begin{array}{r} 101011001 \\ \text{and}\quad 11101 \\ \hline 11001 \end{array}$$

$$\begin{array}{r} 101010110 \\ \text{or}\quad 111011 \\ \hline 101111111 \end{array}$$

$$\begin{array}{r} 1100011 \\ \text{xor } 110110 \\ \hline 1010101 \end{array}$$

We notice that all of the operators perform the operation to each bit position independently of the rest of the positions in the numbers.

2.6 Binary to Decimal Conversion

How do we go from binary numbers to decimal and back? We'll now examine how to take binary numbers and convert them to their decimal notation. This works exactly like the decimal decomposition we learned at the beginning of class.

And in fact this works to convert a number represented in any base to decimal.

$$\begin{aligned} 101011001_2 &= 2^9 &&+2^7 + 2^5 &&+2^4 + 2^0 \\ &= 512 &&+127 + 32 &&+16 + 1 \\ &= 689 \end{aligned}$$

2.7 Decimal to Binary Conversion

Now, let's look at the opposite conversion, converting decimal numbers to binary. This will use successive division and subtraction as opposed to addition in multiplication. We will successively divide by two, if the remainder is non-zero we'll keep the bit at the current position. We'll continue until we can no longer divide by two. Again, this works for any base, not just base two.

Initial Value	Result	Remainder
267	133	1
133	66	1
66	33	0
33	16	1
16	8	0
8	4	0
4	2	0
2	1	0
1	0	1

$$\mathbf{267_{10} = 100001011_2}$$

2.8 Conclusion

Today, we covered two very important number representations. In particular, decimal notation that we're used to seeing in everyday life and binary notation that is useful for understanding how computers manipulate numbers. We covered binary addition, multiplication, and three bitwise arithmetic operations (and, or, and xor). Then we showed how we can convert between binary and decimal notation for numbers. Next lecture, we'll review what we've learned about binary numbers and introduce Hexadecimal numbers, another important number representation in computer science.

3

Arrays

Topics

- *Motivation*
- *Definition*
- *Creating arrays*
- *Indexing*
- *Array length*
- *Default initialization*
- *Bounds checking*
- *Empty arrays*
- *Enhanced for loop*
- *Exchanging and shuffling*
- *Exercises*
- *References*

3.1 Motivation

Consider this snippet of code:

```
if      (day ==  0) System.out.println("Monday");
else if (day ==  1) System.out.println("Tuesday");
else if (day ==  2) System.out.println("Wednesday");
else if (day ==  3) System.out.println("Thursday");
else if (day ==  4) System.out.println("Friday");
else if (day ==  5) System.out.println("Saturday");
else if (day ==  6) System.out.println("Sunday");
```

Question: What does this code do?

This code prints the day of the week after conditioning on the value of an integer *day*. But this code is repetitive. It

would be useful if we had some way of creating a list of days of the week, and then just specifying which of those days we wanted to print. Something like this:

```
System.out.println(DAYS_OF_WEEK[day]);
```

To achieve this in Java, we need arrays.

3.2 Definition

An *array* is an ordered and fixed-length list of values that are of the same type. We can access data in an array by *indexing*, which means referring to specific values in the array by number. If an array has `n` values, then we think of it as being numbered from `0` to `n-1`:

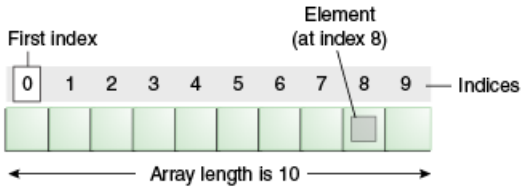

Fig. 1: Diagram of an array (Credit: https://www.geeksforgeeks.org/arrays-in-java/).

To *loop* or *iterate* over an array means that our program accesses every value in the array, typically in order. For example, if we looped over the array in the diagram, that would mean that we looked at the value at the 0th index, then the value at the 1st index, then the value at the 2nd index, and so on.

When we say that the array is "ordered" is that the relationship between an index and its stored value is unchanged (unless we explicitly modify it). If we loop over an unchanged array multiple times, we will always access the same values.

Arrays are *fixed-length*, meaning that after we have created an array, we cannot change its length. We will see in the next chapter [TK: confirm] that `ArrayLists` are an array-like data structure that allows for changing lengths.

Finally, all the values in an array must be of the same type. For example, an array can hold all floating point numbers or all characters or all strings. But an array cannot hold values of different types.

3.3 Creating arrays

The syntax for creating an array in Java has three parts: 1. Array type 2. Array name 3. Either: array size or specific values

For example, this code creates an array of size `n = 10` and fills it with all `0.0` s

```
double[] arr;                    // Declare array
arr = new double[n];             // Initialize the array
for (int i = 0; i < n; i++) {    // Iterate over array
    arr[i] = 0.0;                 // Initialize elements to 0.0
}
```

The key steps are: we first declare and initialize the array. We then loop over the array to initialize specific values. We can also initialize the array at compile time, for example

```
String[] DAYS_OF_WEEK = {
//  Indices:
//  0         1           2             3           4         5           6
    "Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"
};
```

Notice the difference in syntax. When creating an empty array, we must specify a size. When initialize an array at compile time with specific values, the size is implicit in the number of values provided.

Finally, in Java, it is acceptable to move the brackets to directly after the type declaration to directly after the name declaration. For example, these two declarations are equivalent:

```
int arr[];
int[] arr;
```

3.4 Indexing

Consider the array `DAYS_OF_WEEK` from the previous section. We can *index* the array using the following syntax:

```
System.out.println(DAYS_OF_WEEK[3]);  // Prints "Thursday"
```

In Java, array's are said to use *zero-based indexing* because the first element in the array is accessed with the number `0` rather than *1*.

> **Question:** What does `System.out.println(DAYS_OF_WEEK[1]);` print?
>
> **Question:** What does this code do? What number does it print?
>
> ```
> double sum = 0.0;
> double[] arr = { 1, 2, 2, 3, 4, 7, 9 }
> for (int i = 0; i < arr.length; i++) {
>     sum += arr[i];
> }
> System.out.println(sum / arr.length);
> ```

3.5 Array length

As mentioned previously, arrays are *fixed-length*. After you have created an array, it's length is unchangeable. You can access the length of an array `arr[]` with the code `arr.length`.

> **Question:** What does `System.out.println(DAYS_OF_WEEK.length);` print?
>
> **Exercise:** Write a `for` loop to print the days of the week in order (Monday through Sunday) using an array rather than seven `System.out.println` function calls.

3.6 Default initialization

In Java, the default initial values for numeric primitive types is `0` and `false` for the `boolean` type.

> **Exercise:** Consider this code from earlier:

```
double[] arr;
arr = new double[n];
for (int i = 0; i < n; i++) {
    arr[i] = 0.0;
}
```

Rewrite this code to be a single line.

3.7 Bounds checking

Consider this snippet of code.

Question: Where is the bug?

```
int[] arr = new int[100];
for (int i = 0; i <= 100; ++i) {
    System.out.println(arr[i]);
}
```

The issue is that the program attempts to access the value `arr[100]`, while the last element in the array is `arr[99]`.

This kind of bug is called an "off-by-one error" and is so common... well, it has a name. In general, an off-by-one-error is one in which a loop iterates one time too many or too few.

Question: Where is the off-by-one-error?

```
int[] arr = new int[100];
for (int i = 0; i < array.length; i++) {
    arr[i] = i;
}
for (int i = 100; i > 0; --i) {
    System.out.println(arr[i]);
}
```

Exercise: Fill in the missing code in this `for` loop to print the numbers in reverse order, i.e. `5, 4, 3, 2, 1`:

```
int[] arr = { 1, 2, 3, 4, 5 };
for (???) {
    System.out.println(arr[i]);
}
```

3.8 Empty arrays

Question: This code prints five values, one per line, but we never specified which values. What do you think it prints?

```
int[] arr = new int[5];
for (int i = 0; i < arr.length; i++) {
    System.out.println(arr[i]);
}
```

In Java, an unitialized or empty array is given a default value:

- For `int`, `short`, `byte`, or `long`, the default value is `0`.
- For `float` or `double`, the default value is `0.0`.
- For `boolean` values, the default value is `false`.
- For `char`, the default value is the null character `'\u0000'`.

Note that an array can be partially initialized.

Question: What does this code print?

```
char[] alphabet = new char[26];
alphabet[0] = 'a';
alphabet[1] = 'b';
for (int i = 0; i < alphabet.length; i++) {
    System.out.println(alphabet[i]);
}
```

3.9 Enhanced for loop

So far, we have seen how to iterate over arrays by indexing each element with a number:

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
for (int i = 0; i < vowels.length; ++ i) {
    System.out.println(vowels[i]);
}
```

We can perform the same iteration without using indices using an "enhanced `for` loop" or `for-each` loops:

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
for (char item: vowels) {
    System.out.println(item);
}
```

3.10 Exchanging and shuffling

Two common tasks when manipulating arrays are *exchanging two values* and *shuffling* values. (*Sorting* is more complicated and will be address later.)

To exchange to values, consider the following code:

```
double[] arr = { 1.0, 2.0, 3.0, 4.0, 5.0, 6.0 };
int i = 1;
int j = 4;
double tmp = arr[i];
arr[i] = arr[j];
arr[j] = tmp;
```

Exercise: What are the six values in the array, in order?

To shuffle the array, consider the following code:

```
int n = arr.length;
for (int i = 0; i < n; i++) {
    int r = i + (int) (Math.random() * (n-i));
    String tmp = arr[r];
    arr[r] = arr[i];
    arr[i] = tmp;
}
```

Question: What does this code do:

```
for (int i = 0; i < n/2; i++) {
    double tmp = arr[i];
    arr[i] = arr[n-1-i];
    arr[n-i-1] = tmp;
}
```

3.11 Exercises

1. **Write a program that reverses the order of values in an array.**
2. **What is wrong with this code snippet?**

```
int[] arr;
for (int i = 0; i < 10; i++) {
    arr[i] = i;
}
```

3. **Rewrite this snippet using an enhanced `for-each` loop (for now, it is okay to re-define the array):**

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
for (int i = array.length; i >= 0; i--) {
    char letter = vowels[i];
    System.out.println(letter);
}
```

5. **Write a program that uses `for` loops to print the following pattern:**

```
1********

12*******

123******

1234*****

12345****

123456***

1234567**

12345678*

123456789
```

4. **Write a program** `HowMany.java` **that takes an arbitrary number of command line arguments and prints how many there are.**

3.12 References

- Computer Science: An Interdisciplinary Approach, Robert Sedgewick and Kevin Wayne.

4

ArrayLists

Topics

- *Motivation*
- *ArrayLists*
- *In-class assignment*
- *References*

4.1 Motivation

4.2 ArrayLists

A _collection_ is a group of objects. Today, we'll be looking at a very useful collection, the `ArrayList`. A _list_ is an ordered collection, and an `ArrayList` is one type of list.

Create a class NameTracker and follow along in it.

Before we can use an ArrayList, we have to import it:

```
import java.util.ArrayList;
```

Next, we call the constructor; but we have to declare the type of object the `ArrayList` is going to hold. This is how you create a new `ArrayList` holding `String` objects.

```
ArrayList<String> names = new ArrayList<String>();
```

Notice the word "String" in angle brackets: "". This is the Java syntax for constructing an ArrayList of String objects.

We can add a new String to names using the `add()` method.

```
names.add("Ana");
```

> **Question:** Exercise: Write a program that asks the user for some names and then stores them in an ArrayList. Here is an example program:

```
Please give me some names:
Sam
Alecia
Trey
Enrique
Dave
Your name(s) are saved!
```

We can see how many objects are in our ArrayList using the size() method.

```
System.out.println(names.size()); // 5
```

Exercise: Modify your program to notify the user how many words they have added.

```
Please give me some names:
Mary
Judah
Your 2 name(s) are saved!
```

Remember how the `String.charAt()` method returns the `char` at a particular index? We can do the same with names. Just call `get()`:

```
names.add("Noah");
names.add("Jeremiah");
names.add("Ezekiel");
System.out.println(names.get(2)); // "Ezekiel"
```

Exercise: Update your program to repeat the names back to the user in reverse order. Your solution should use a for loop and the size() method. For example:

```
Please give me some names:
Ying
Jordan

Your 2 name(s) are saved! They are:
Jordan
Ying
```

Finally, we can ask our names `ArrayList` whether or not it has a particular string.

```
names.add("Veer");
System.out.println(names.contains("Veer")); // true
```

Exercise: Update your program to check if a name was input by the user. For example:

Please give me some names: Ying Jordan

Search for a name: Ying Yes!

An `ArrayList` can hold any type of object! For example, here is a constructor for an `ArrayList` holding an instance of a `Person` class:

```
ArrayList<Person> people = new ArrayList<Person>();
```

where `Person` is defined as

```
public class Person {
```

(continues on next page)

(continued from previous page)

```
    String name;
    int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

> **Exercise:** Modify our program to save the user's input names as Person instances. Rather than storing String objects in the ArrayList, store Person objects by constructing them with the input name. You'll need to use the Person constructor to get a Person instance!

4.3 In-class assignment

Write a class BlueBook that tells the user the price of their car, depending on the make, model, and year. You should use Car.java and the stencil file provided, BlueBook.java.

Your program depends on what cars your BlueBook supports, but here is an example program:

```
What is your car's make?
Toyota
What is your Toyota's model?
Corolla
What is your Toyota Corolla's year?
1999
```

Your 1999 Toyota Corolla is worth $2000.

> **Bonus exercise:** Notify the user if the car is not in your BlueBook.
>
> **Bonus exercise:** Clean up main by putting your code for creating the ArrayList in a separate method. What type should the method return?
>
> **Bonus exercise:** If the car is not in the BlueBook, ask the user to input the relevant data, construct a new Car instance, add it to your ArrayList.

4.4 References

1. https://github.com/accesscode-2-1/unit-0/blob/master/lessons/week-3/2015-03-24_arraylists.md