Day 1: Data Types

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate).

Data Types

Data types define and restrict what type values can be stored in a variable, as well as set the rules for what types of operations can be performed on it.

Primitive Data Types

Java has 8 primitive data types: *byte*, *short*, *int*, *long*, *float*, *double*, *boolean*, and *char*. For most challenges, you'll only need to concern yourselves with ints (e.g.: , , , etc.) and doubles (e.g.: , , , etc.). Another important data type we mentioned yesterday is the String class, whose objects are immutable strings of characters.

Scanner

Yesterday, we discussed Scanner's *next*, *nextLine*, *hasNext*, and *hasNextLine* methods. Scanner also has *readNext*and *hasNext* methods for different data types, which is very helpful when you know exactly what type of input you'll be reading. The *next* methods scan for *tokens* (you can think of this as a word), and the *nextLine* methods reads from the Scanner's current location until the beginning of the next line. For example, *nextInt()* will scan the next token of input as an *int*, and *nextDouble()* will scan the next token of input as a *double*. *You should only ever use scanner object for your entire program.*

Each line of multi-line input contains an invisible separator indicating that the end of a line of input has been reached. When you use Scanner functions that read tokens (e.g.: *next()*, *nextInt()*, etc.), the Scanner reads and returns the next token. When you read an entire line (i.e.: *readLine()*), it reads from the current position until the beginning of the next line. Because of this, a call to *nextLine()* may return an empty string if there are no characters between the end of the last read and the beginning of the next line. For example, given the following input:

```
a b cd efg
```

The breakdown below shows how a certain sequence of calls to a Scanner object, , will read the above input:

1. A call to `scan.next();` returns the next token, `a`.
2. A call to `scan.next();` returns the next token, `b`.

3. A call to `scan.nextLine();` returns the next token, `c`. It's important to note that the scanner returns a space *and* a letter, because it's reading from the end of the last token until the beginning of the next line.
4. A call to `scan.nextLine();` returns the contents of the next line, `d e`.
5. A call to `scan.next();` returns the next token, `f`.
6. A call to `scan.nextLine();` returns everything after `f` until the beginning of the next line; because there are no characters there, it returns *an empty String*.
7. A call to `scan.nextLine();` returns `g`.

Note: *You will struggle with this challenge if you did not review this section. You must understand what happens when you switch between reading a token (single word) of input and reading an entire line of input to successfully complete this challenge.*

Additive Operator

The `+` operator is used for mathematical addition and String concatenation (i.e.: combining two Strings into one new String). If you add the contents of two variables together (e.g.: `a + b`), you can assign their result to another variable using the *assignment operator* (`=`). You can also pass the result to a function instead of assigning it to a variable; for example, if and , `System.out.println(a + b);` will print `3` on a new line.

C++

You may find this information helpful when completing this challenge in C++.

To consume the whitespace or newline between the end of a token and the beginning of the next line:

```
if (getline(cin >> ws, s2)) { // eat whitespace
    getline(cin, s2);}
```

where `s2` is a string. In addition, you can specify the *scale* of floating-point output with the following code:

```
#include <iostream>#include <iomanip>
using namespace std;int main(int argc, char *argv[]) {
    double pi = 3.14159;

    // Let's say we wanted to scale this to 2 decimal places:
    cout << fixed << setprecision(2) << pi << endl;
    printf("%.2f", pi);}
```

which produces this output:

```
3.14

3.14
```

Day 2: Operators

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate).

Operators

These allow you to perform certain operations on your data. There are 3 basic types:

1. *Unary*: operates on 1 operand
2. *Binary*: operates on 2 operands
3. *Ternary*: operates on 3 operands

Arithmetic Operators

The binary operators used for arithmetic are as follows:

- `+`: Additive
- `-`: Subtraction
- `*`: Multiplication
- `/`: Division
- `%`: Remainder (*modulo*)

Additional Operators

- `+`: A binary operator used for String concatenation
- `++`: This unary operator is used to *preincrement* (increment by 1 before use) when prepended to a variable name or *postincrement* (increment by 1 after use) when appended to a variable.
- `--`: This unary operator is used to *predecrement* (decrement by 1 before use) when prepended to a variable name or *postdecrement* (decrement by 1 after use) when appended to a variable.

- `!`: This unary operator means *not* (negation). It's used before a variable or logical expression that evaluates to true or false.
- `==`: This binary operator is used to check the *equality* of 2 primitives.
- `!=`: This binary operator is used to check the *inequality* of 2 primitives.
- `<`, `>`, `<=`, `>=`: These are the respective binary operators for *less than*, *greater than*, *less than or equal to*, and *greater than or equal to*, and are used to compare two operands.
- `&&`, `||`: These are the respective binary operators used to perform *logical AND* and *logical OR* operations on two boolean (i.e.: true or false) statements.
- `? :` This ternary operator is used for simple conditional statements (i.e.: if ? then : else).

Day 3: Intro to Conditional Statements

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate).

Boolean

A logical statement that evaluates to *true* or *false*. In some languages, *true* is interchangeable with the number and *false* is interchangeable with the number .

Conditional Statements

These are a way of programming different workflows depending on some boolean condition.
The *if-else*statement is probably the most widely used conditional in programming, and its workflow is demonstrated below:

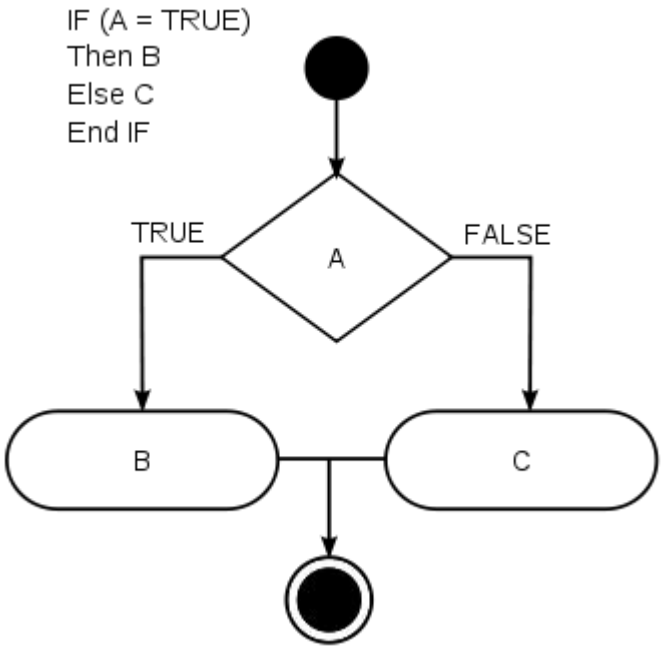

Image Source: Wikipedia(Conditional Statements)

The basic syntax used by Java (and many other languages) is:

```
if(condition) {

    // do this if 'condition' is true}else {

    // do this if 'condition' is false}
```

where is a boolean statement that evaluates to true or false. You can also use an `if` *without* an `else`, or follow an `if(condition)` with `else if(secondCondition)` if you have a second condition that only need be checked when is false. If the `if` (or `else if`) condition evaluates to true, any other sequential statements connected to it (i.e.: `else` or an additional `else if`) *will not execute.*

Logical Operators

Customize your condition checks by using logical operators. Here are the three to know:

- `||` is the *OR* operator, also known as *logical disjunction*.
- `&&` is the *AND* operator, also known as *logical conjunction*.
- `!` is the *NOT* operator, also known as *negation*.

Here are some usage examples:

```
// if A is true and B is true, then Cif(A && B){

    C;    }

    // if A is true or B is true, then Cif(A || B){

    C;}

    // if A is false (i.e.: not true), then Bif(!A){
```

```
    B;}
```

Another great operator is the *ternary* operator for conditional statements (`? :`). Let's say we have a variable, , and a condition, . If the condition is true, we want to be assigned the value of ; if condition is false, we want to be assigned the value of . We can write this with the following simple statement:

```
v = c ? a : b;
```

In other words, you can read `c ? a : b` as "if is true, then ; otherwise, ". Whichever value is chosen by the statement is then assigned to .

Switch Statement

This is a great control structure for when your control flow depends on a number of *known values*. Let's say we have a variable, , whose possible values are , , , and each value has an action to perform (which we will call some variant of). We can *switch* between actions with the following code:

```
switch (condition) {
    case val0:      behavior0;
                    break;
    case val1:      behavior1;
                    break;
    case val2:      behavior2;
                    break;
    default:        behavior;
                    break;}
```

Note: Unless you include `break;` at the end of each case statement, the statements will execute sequentially. Also, while it's good practice to include a `default:` case (even if it's just to print an error message), it's not strictly necessary.

Day 4: Class vs. Instance

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate).

Class

A blueprint defining the charactaristics and behaviors of an object of that class type. Class names should be written in CamelCase, starting with a *capital* letter.

```
class MyClass{

    ...}
```

Each class has two types of variables: class variables and instance variables; class variables point to the same (static) variable across all instances of a class, and instance variables have distinct values that vary from instance to instance.

Class Constructor

Creates an instance of a class (i.e.: calling the Dog constructor creates an instance of Dog). A class can have one or more constructors that build different versions of the same type of object. A constructor with no parameters is called a default constructor; it creates an object with default initial values specified by the programmer. A constructor that takes one or more parameters (i.e.: values in parentheses) is called a *parameterized constructor*. Many languages allow you to have multiple constructors, provided that each constructor takes different types of parameters; these are called overloaded constructors.

```
class Dog{ // class name

    static String unnamed = "I need a name!"; // class variable

    int weight; // instance variable

    String name; // instance variable

    String coatColor; // instance variable


    Dog(){ // default constructor

        this.weight = 0;

        this.name = unnamed;

        this.coatColor = "none";
```

```
    }
    Dog(int weight, String color){ // parameterized constructor
        // initialize instance variables
        this.weight = weight; // assign parameter's value to instance variable
        this.name = unnamed;
        this.coatColor = color;
    }
    Dog(String dogName, String color){ // overloaded parameterized constructor
        // initialize instance variables
        this.weight = 0;
        this.name = dogName;
        this.coatColor = color;
    }}
```

Method

A sort of named procedure associated with a class that performs a predefined action. In the sample code below, *returnType* will either be a data type or if no value need be returned. Like a constructor, a method can have or more parameters.

```
returnType methodName(parameterOne, ..., parameterN){
    ...
    return variableOfReturnType; // no return statement if void}
```

Most classes will have methods called *getters* and *setters* that get (return) or set the values of its instance variables. Standard getter/setter syntax:

```
class MyClass{
    dataType instanceVariable;
    ...
    void setInstanceVariable(int value){
```

```
        this.instanceVariable = value;
    }
    dataType getInstanceVariable(){
        return instanceVariable;
    }}
```

Structuring code this way is a means of managing how the instance variable is accessed and/or modified.

Parameter

A parenthetical variable in a function or constructor declaration (e.g.: in `int methodOne(int x)`, the parameter is `int x`).

Argument

The actual value of a parameter (e.g.: in `methodOne(5)`, the argument passed as variable `x` is `5`).

Day 5: Loops

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate). As you code more, you may see these loops implemented in different ways than are shown here.

For Loop

This is an iterative loop that is widely used. The basic syntax is as follows:

```
for (initialization; termination; increment) {
   // ...}
```

The *initialization* component is the starting point in your iteration, and your code for this section will typically be `int i = 0`. When we declare and initialize `int i` in the loop like this, we are creating a *temporary variable* that exists only inside this loop for the purposes of iterating

through the loop; once we finish iterating and exit (or *break*) the loop, is deleted and can be declared elsewhere in our program.

The *termination* component is the condition which, once met, you would like to exit (or *break*) the loop and proceed to the next line in your code. This is the ending point for your loop, and is typically written as `i < endValue`, where is the variable from the initialization section and is some variable holding the stopping point for your iteration.

The *increment* component is executed each time the end of the code inside the loop's brackets is reached, and should generally be some modification on the initialization variable that brings it closer to the termination variable. This will typically be `i++`. The `++` operator is also called the post-increment operator, and it will increment a variable by after a line executes (for more detail and an example, see the *While* section).

To recap, this sample code:

```
int endOfRange = 4;for(int i = 0; i < endOfRange; i++){

    System.out.println(i);}
```

produces this output:

```
0

1

2

3
```

While Loop

This type of loop requires a single boolean condition and continues looping as long as that condition continues to be true. Each time the the end of the loop is reached, it loops back to the top and checks if the condition is still true. If it's true, the loop will run again; if it's false, then the program will skip over the loop and continue executing the rest of the code.

Much like in the *For* section, the code below prints the numbers through . Notice that we are using the *post-increment* operator on :

```
int min = 0;int max = 4;while(min < max){

    System.out.println(min++);}
```

Once , the boolean condition () evaluates to false and the loop is broken. The line `System.out.println(min++);` is a compact way of writing:

```
System.out.println(min);min = min + 1;
```

Do-While Loop

This is a variation on the *While* loop where the condition is checked at the end of the brackets. Because of this, the content between the brackets is guaranteed to always be executed at least once:

```
do{
    // this will execute once
    // it will execute again each time while(condition) is true} while(condition);
```

Unlabeled Break

You may recall *break;* from our previous discussion of *Switch Statements*. It will break you out of a loop even if the loop's termination condition still holds true.

Day 6: Let's Review

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate).

Strings and Characters

As we've mentioned previously, a String is a sequence of characters. In the same way that words inside double quotes signify a String, a single letter inside single quotes signifies a character. Each character has an ASCII value associated with it, which is essentially a numeric identifier. The code below creates a char variable with the value , and then prints its ASCII value.

```
char myChar = 'c'; // create char cSystem.out.println("The ASCII value of " + myChar + " is:
 " + (int) myChar);
```

Output:

```
The ASCII value of c is: 99
```

Observe the `(int)` before the variable name in the code above. This is called *explicit casting*, which is a method of representing one thing as another. Putting a data type inside parentheses right before a variable is essentially saying: "The next thing after this should be represented as this data type". Casting only works for certain types of relationships, such as between primitives or objects that inherit from another class.

To break a String down into its component characters, you can use the String.toCharArray method. For example, this code:

```
String myString = "This is String example.";char[] myCharArray = myString.toCharArray();for(int i = 0; i < myString.length(); i++){

    // Print each sequential character on the same line

    System.out.print(myCharArray[i]); }// Print a newlineSystem.out.println();
```

produces this output:

```
This is String example.
```

Notice that we were able to simulate printing *myString* by instead printing each individual character in the character array, *myCharArray*, created from *myString*.

Day 7: Arrays

Terms you'll find helpful in completing today's challenge are outlined below, along with sample Java code (where appropriate).

Data Structures

A way of organizing data that enables efficient storage, retrieval, and use.

Arrays

A type of data structure that stores elements of the same type (generally). It's important to note that you'll often see arrays referred to as in documentation, but the variable names you use when coding should be descriptive and begin with *lowercase* letters.

You can think of an array, , of size as a contiguous block of cells sequentially indexed from to which serve as containers for elements of the array's declared data type. To store an element, , in some index of array , use the syntax `A[i]` and treat it as you would any other variable (i.e., `A[i] = value;`). For example, the following code:

```
// the number of elements we want to holdfinal int _arraySize = 4;

// our array declarationString[] stringArray = new String[_arraySize];

for(int i = 0; i < _arraySize; i++) {

    // assign value to index i

    stringArray[i] = "This is stored in index " + i;


    // print value saved in index i

    System.out.println(stringArray[i]); }
```

saves and then prints the values listed below in their respective indices of :

```
This is stored in index 0

This is stored in index 1

This is stored in index 2

This is stored in index 3
```

Most languages also have a *method*, *attribute*, or *member* that allows you to retrieve the size of an array. In Java, arrays have a attribute; in other words, you can get the length of some array, `arrayName`, by using the `arrayName.length` syntax.

Note: The *final* keyword used in the code above is a means of protecting the variable's value by locking it to its initialized value. Any attempt to reassign (overwrite) the value of a *final* variable will generate an error.

Note on Arrays in C++

If you want to create an array whose size is unknown at compile time (i.e., being read as input), you need to create a *pointer* to whatever data type you'll be declaring your array as

(e.g., *char*, *int*, *double*, etc.). Then you must use the new operator to set aside the space you need for your array. The example below shows how to create an array of type *DataType* and unknown size *n* that is read from stdin.

```
// array sizeint n; cin >> n;

// create array of unknown size nDataType* arrayName = new DataType[n];
```