4 - STRINGS

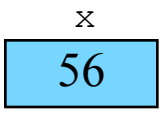
4.1 Introduction

We have learned that there are eight primitive types in Java. We are now going to study an object type: The `String` type. The number of object types in Java is unlimited because object types can be defined by the programmer.

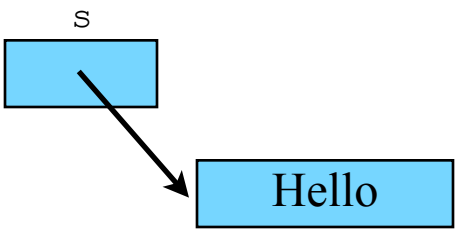
The `String` type has been predefined by Java programmers in the same way that we will learn to define our own types.

We declare a string object variable in a similar way as a primitive. If the string variable is not initialized, it is declared but *undefined*. Once the variable has been assigned a value, it points to this value. *A primitive variable contains the value that is assigned to it. An object variable contains a reference to where the value is stored.* The following illustrates the difference:

```
int x = 56;
```




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



The variable `s` does not contain the word `Hello`. It contains a reference that points to the memory location where the word `Hello` is stored. Java needs to do this because the length of the string is not known until assignment time. Every time a value is assigned to the variable `s` it must be stored in a new memory location, and `s` must point to this new location.

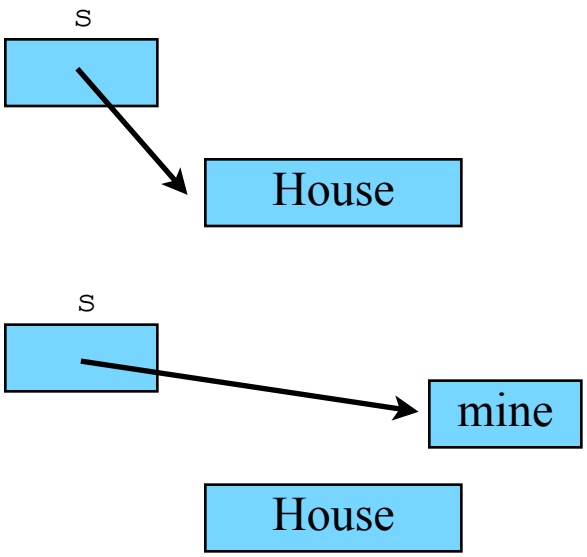
Because `s` contains a reference to a string value, we say that `s` is an example of a *reference type*.

Once a string is created, its value cannot be changed. We say that a string is an *immutable* object. What this means is that we do not have direct access to the memory location where the string resides to change its contents. However, the reference `s` can be changed.

The following program fragment

```
String s = "House";
s = "mine";
```

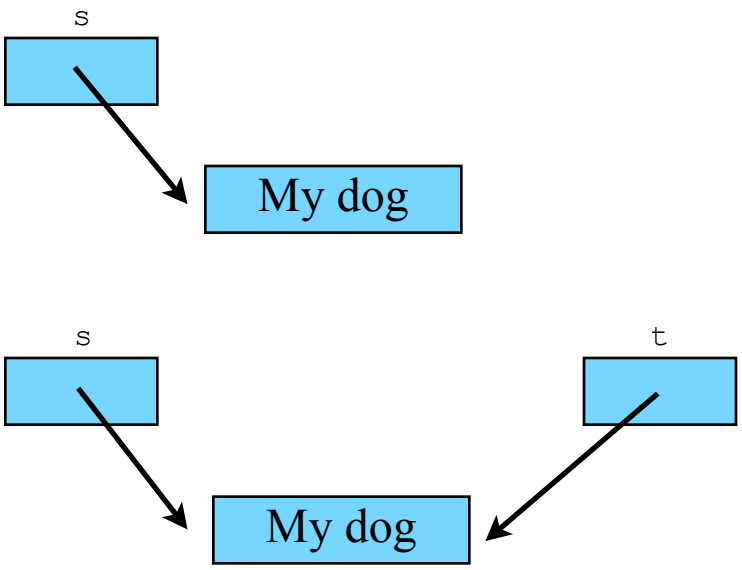
has the following effect in memory



In the declaration and initialization statement, `s` references the string `House`. Then, in the assignment statement, `s` references the string `mine` and the string `House` is no longer accessible, even though it is still somewhere in memory. This memory is now wasted since it is not being referenced by any variables. Memory will be recovered by the process of *garbage collection* which runs in the background.

We can assign one string variable to another string variable. Like primitives, a copy is performed. But the string variables contain references. Therefore, what is copied is not the value but the reference. This is very different from primitives. The following program fragment illustrates this process:

```
String s = "My dog";
String t = s;
```



The operating system has the responsibility of managing memory space so that memory is given to a program that places a request. Everytime we declare a variable in a program, we are requesting memory space which the operating system grants if available. Since many programs could be requesting memory space simultaneously, it is the operating system's responsibility to make sure that programs do not use each other's spaces.[1]

4.1 Exercises

1. Draw diagrams like those above to illustrate the result of executing the following statements:

```
a) int a = 1;
   String b = "How are you?";


b) String s = "first";
   String b = "second";

c) String a = "one";
```

[1] In the case of multi-threading two concurrent processes can and do share data memory spaces. This is a powerful tool to communicate between processes.

```
   String b = a;
   a = "two";
```

d)
```
   String a = "uno";
   String b = "dos";
   a = b;
   b = "tres";
```

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

```
   String c = "cat";
   String d = "dog";
   String s = c;
   c = d;
   d = s;
   System.out.println("c is " + c);
   System.out.println("d is " + d);
```

3. True/False

a. A string variable can be declared without initialization
b. A string variable is a reference type
c. The value referenced by a string variable can be changed by a Java program
d. The reference that a string variable contains can be changed by a Java program
e. Two string variables can contain the same reference, i.e., they can point to the same memory location
f. Garbage collection happens when a memory location is not being referenced by any variable
g. It is the responsibility of the operating system to give memory to a program when the program requests it
h. It is the responsibility of the Java programmer to make sure that two programs do not share the same memory space

4.2 String Methods

Once a string variable has been declared and initialized, there are many methods that become available. All these methods operate on the variable. We will start with the most useful and common string methods:

- `substring()`
- `charAt()`
- `indexOf()`
- `length()`
- `equals()`
- `compareTo()`
- `compareToIgnoreCase()`

4.2.1 substring()

This method returns a portion of a string. In the following fragment, the area code is removed from a phone number and then printed:

```
String phone = "416-393-1722";
String noArea = phone.substring(4, 12);
System.out.println(noArea);
```

Note the following:

1. The call to the method is done by writing the variable followed by a period and then followed by the method name. The method will have an effect on the variable that makes the call
2. The `substring()` method takes two parameters:
   a. The starting position (where 0 is the first character of the string)
   b. The ending position (where 1 is the first character of the string)

The `substring()` method also works with only one parameter. In that case, the portion returned is from the specified position to the end of the string. For example,

```
String phone = "416-393-1722";
System.out.println(phone.substring(8));
```

returns and prints the last characters of `phone` starting at position 8, where 0 is the first character of `phone`

4.2.2 charAt()

This method returns the character specified by a position. For example:

```
String phone = "416-393-1722";
System.out.println(phone.charAt(2));
```

returns the character at position 2. This is `6`. The value returned is a character, not a string.

4.2.3 indexOf()

Looks for a string inside a string and returns its position. For example,

```
String phone = "416-393-1722";
System.out.println(phone.indexOf("393"));
```

results in `4` because the string `393` is in position 4 of the string `416-393-1722`. If the string cannot be found, then `-1` is returned.

4.2.4 length()

This method returns the length of a string. For example,

```
String phone = "416-393-1722";
System.out.println(phone.length());
```

will print a `12` because the string contains 12 characters.

4.2.5 equals(), compareTo(), compareToIgnoreCase()

When comparing the contents of two String variables in Java, we need to use *comparison methods.* We cannot use logical operators such as ==, >, or < because these operators compare memory addresses.

String Ordering

When comparing strings with strings the ordering follows the ASCII table:

a) Capitals have a lower value than lower case letters
b) Single numbers ascend as in single digits
c) Numbers have a lower value than letters

The following strings are in strictly ascending order (all these are `String` type),

```
1
14
156
3
36
F
Fred
L
c
dome
```

The comparison

```
if (name == last)
```

will <u>not</u> check if the string that `name` points to is the same as the string that `last` points to. Rather it will compare if the variables `name` and `last` point to the same memory location.

Some Comparison Methods

The comparison

```
if (name.equals("Hello"))
```

will check if the contents of the variable `name` is the word `Hello` and nothing else.

Consider the next code segment,

```
String lastName = "Gonzalez";
String initial = "E";

if (lastName.compareTo(initial) < 0) {
    Stdout.println(lastName);
}
```

This previous code will compare the contents of the variable `lastName` against the contents of the variable `initial`. If `lastName` contains a value lower in alphabet than `initial`, then the method `compareTo()` will give a number smaller than 0. If `lastName` contains a larger value than `initial` then `compareTo()` will give a value larger than 0. If they are equal, `compareTo()` will give 0.

The expression `lastName.compareToIgnoreCase("gonzalez") == 0` is true since even though the word `gonzalez` is different from `Gonzalez`. The method ignores the case of the strings being compared.

A common error when comparing strings in Java is the following

```
String lastName = "Gonzalez";
String initial = "E";

if (lastName < initial) {
    Stdout.println(lastName);
}
```

The name `Gonzalez` will be printed if and only if the ***memory location*** of the string that `lastName` points to is smaller than the ***memory location*** of the string that `initial` points to. At this point we are not dealing with memory allocation ourselves, but this will happen later in programming.

Note: We need to be careful not to confuse `String` type with `char` type. Single characters can be compared with the logical operators because characters are considered to be a numeric type.

4.2.6 Method summary

We must be very careful to ensure that the numbers we put in the calls to the string methods are valid. If we put a number which goes beyond the length of the string, a *run-time* error will occur. The standard error message for this sort of problem is `String index out of range`

Note: In the summary below, the word **`this`** *refers to the string variable that makes use of the method. For example, in a command like* `text.substring(i)` **`this`** *refers to the variable* `text`*. Also, the type given at the beginning of the line is the type that the method returns. For example* **`char`** `charAt(int index)` *means that the the method* `charAt()` *returns a character. The methods in* ***boldface*** *are the most widely used string methods.*

Method	Description
`char charAt(int index)`	Return character at position index of **this**
`boolean equals(Object <object>)`	Return equality of **this** and object
`int indexOf(String <string>)`	Return position of a string in **this**
`int length(String <string>)`	Return length of **this**
`String substring(int <startIndex>, int <endIndex>)` `Note: endIndex is calculated with 1 as first position of string`	Return substring of **this** starting at startIndex and ending at endIndex (note that endIndex is calculated with 1 as first position of string)
`String substring(int <startIndex>)`	Return substring of **this** starting at startIndex to the end of this. Overloaded.
`boolean endsWith(String <suffix>)`	Return whether or not **this** ends in suffix
`boolean equalsIgnoreCase(String <string>)`	Return equality of **this** and string ignoring case
`String replace(Char <oldChar>, Char <newChar>)`	Replace every occurrence of oldChar with newChar in **this**
`String toLowerCase()`	Return **this** in lower case
`String toUpperCase()`	Return **this** in upper case
`String trim()`	Return **this** with leading and trailing blanks removed
`static String valueOf(int <value>)`	Return value as a String. Method can be overloaded to other number types

Further descriptions of String methods can be found at:

http://docs.oracle.com/javase/7/docs/api/java/lang/String.html

4.2 Exercises

1. Write a program that asks the user to enter a phone number with dashes in the standard pattern `xxx-xxx-xxxx`. Then output the phone number without its area code, i.e., skip the first 3 numbers and the dash in the fourth position. Exit the program when the user types the phone number `000-000-0000`

2. Write a program that asks the user for two words of any length. The program prints `true` if the first word is contained in the second. Otherwise it prints `false`.

3. Write a program that asks the user for the first name and the last name of a person separated by a space. Both names are stored in a single `String` object. Your program then splits the two words, stores them in two separate `String` objects and prints them in separate lines. Do not store the space that separates the words in any of the two variables. For example,

```
Enter name: John Smith
John
Smith
```

4. Write a program that asks the user for a word of any length. The program reverses the word and prints it out. For example,

```
Enter word: Pal
Reverse is: laP
```

5. Write a Java program that keeps track of the number of characters in a series of 10 words that you enter. Then compute the average word length.

6. Write a Java program that determines if a word is a palindrome. A palindrome is a word that when reversed results in the same word. Examples of palindromes are: `madam, anna, mom.` If we ignore spaces, the following are palindromes: `poor dan is in a droop, star comedy by democrats`

7. Internet protocol (IP) addresses are of the form

```
ddd.ddd.ddd.ddd
```

where `d` is a digit and each pattern `ddd` is be in the range [0, 255]. Here are some examples of valid IP addresses:

```
192.168.2.11
175.30.1.127
80.200.15.64
```

Part 1. For part of the exercise we will not worry about the numbers falling in the [0, 255] range. For now, we need to write a program that determines if a given IP-address has the dots in the correct placement. If so, the program prints the word `correct`. Otherwise the program prints the word `incorrect`. All these conditions must be met for a given address to be have the correct dot placement:

- The length is at least 7 and at most 15
- Neither the first nor the last character is a dot
- There are exactly three dots
- No two dots are beside each other

We can use the following cases to test our program

test	output
`1.1.23`	`incorrect`
`123.199.271.1234`	`incorrect`
`.13.225.114.1`	`incorrect`
`45.255.11.92.`	`incorrect`
`..1`	`incorrect`
`.13..`	`incorrect`
`...255`	`incorrect`
`192.168.2.1`	`correct`

test	output
`90.12.14.255`	`correct`
`255.255.255.255`	`correct`
`0.0.0.0`	`correct`

Part 2. Now that we have the code that correctly detects if a string has the correct IP-address format, we can determine that the characters contained between the dots are numbers in the range [0, 255].

To check that a character is a digit we can use the method

```
Character.isDigit(<character>)
```

which returns true if `character` is a digit from 0 to 9, or `false` otherwise. Here are some examples:

method call	evaluates to
`Character.isDigit('G')`	`FALSE`
`Character.isDigit('6')`	`TRUE`
`String phone = "416-463-2283"` `Character.isDigit(phone.charAt(3))`	`FALSE`
`String phone = "416-463-2283"` `Character.isDigit(phone.charAt(11))`	`TRUE`

Once we know that a set of characters is made up of only digits, we can check that they are in the range [0, 255]. The method

```
Integer.parseInt(<String>)
```

converts a string to an integer. If the given string contains something other than digits, the method will crash the program. This is the reason we need to check first that the passed string is made up of only numbers.

method call	evaluates to
`Integer.parseInt("3207")`	`3207`
`Integer.parseInt("A23")`	`program crash`

Here is a suggested algorithm in pseduo-code that checks each portion of numbers between the dots separately. As soon as a character is found that is not a digit, the algorithm ends and prints "not an ip-address".

If you use this algorithm, simply copy and paste it to the Java editor and then replace the English commands with Java code.

algorithm

```
let address be the ip-address to check
if address has the correct format as specified in Part 1 {
    let s = address plus an extra dot tagged on at the end   // ddd.ddd.ddd.ddd.
    let goAhead = true

    while (s is not equal to a null string and goAhead) {
        let t = substring from the beginning of s to the first dot,
                not including the dot

        for (every character in t and goAhead) {
            if (current character is not a digit) then goAhead = false
        }
        if (goAhead) {
            if (t is not in the range [0, 255]) then goAhead = false
        }
        let s = substring from the first character after the first dot of s
                to the end of s
    }

    if (goAhead) print valid ip-address
    else print not an ip-address
}
else {
    System.out.println("incorrect format");
}
```

(test cases below)

test	output
175.192.400.200	not an ip-address
192.168.2.b	not an ip-address
10.ff.20.19	not an ip-address
10.175.200.225	valid ip-address
225.100.0.1	valid ip-address
255.255.255.255	valid ip-address
0.0.0.0	valid ip-address

Part 3. In this last part of the exercise we need to integrate the programs of parts 1 and 2 so that an ip-address can be checked in a single program. The overall structure of our each of our programs has been

```
let address be the ip-address to check
if address successfully passes all format tests {
    print “correct”
}
else {
    print “incorrect”
}

if address passes the test of part 1 {
    if  all characters of address are digits and the numbers are in correct range {
        print “valid ip-address”
    }
    else {
        print “not an ip-address”
    }
}
```

We can integrate the code of both parts in two different ways:

a) We can embed the code of part 2 within the innermost `if` statement of part 1 so that part 2 is only done if part 1 is successful. The structure is:

```
let address be the ip-address to check
```

```
if address successfully passes all format tests {
    if  all characters of address are digits and the numbers are in correct range {
        print “valid ip-address”
    }
    else {
        print “not an ip-address”
    }
}
else {
    print “incorrect”
}
```

b) Alternatively we can keep the two parts without embedding one inside the other. Instead, we initialize the variable `goAhead` to `true` and set it to `false` any time that a test from part 1 fails. Then, we check the value of `goAhead` prior to entering into part 2. The structure is:

```
let address be the ip-address to check
let goAhead be true
if address successfully passes all format tests {
    print “correct”
}
else {
    print “incorrect”
    set goAhead to false
}

if goAhead {
    if  all characters of address are digits and the numbers are in correct range {
        print “valid ip-address”
    }
    else {
        print “not an ip-address”
    }
}
```

4.2 More Exercises on Strings

1. Write a Java program that asks the user for a first name and a last name and puts them both on a single variable. The program prints out the last name followed by a comma followed by the initial of the first name, followed by a period. For example:

```
Name: Carol Queen
Queen, C.
```

2. Write a Java program that gets a word from the user and prints each letter of the word on a separate line. For example,

```
Word: Car
C
a
r
```

3. Write a Java program that gets a word from the user and prints the ASCII value of each letter of the word. For example,

```
Word: Tom
84
111
109
```

4. Write a Java program that gets an ASCII number from 32 to 127 and prints the corresponding character. For example,

```
ASCII number: 121
y
```

5. Write a Java program that asks the user for a word and prints the ASCII value of each letter as well as the next higher ASCII value. For example,

```
Word: Desk
68      69
101     102
115     116
107     108
```

6. Write a Java program that asks the user for a character, changes the character to its corresponding ASCII value, adds 2 to the ASCII value, then converts back to a character. The final character is printed. For example,

```
Character: F
H
```

7. Write a Java program that uses a loop to ask the user for 5 words. The program prints the total number of characters that all the words have together. For example,

```
Word: Computer
Word: Cat
Word: Floor
Word: Rain
Word: Surprise
28
```

8. Write a Java program that asks the user for a word and prints out the word with its characters changed to the next character in the alphabet. For example,

```
Word: Summer
Tvnnfs
```

9. Write a Java program with the class name `AlphabeticalNames` that asks the user for two names. The program prints them out in alphabetical order ascending.

10. Write a Java program with the class name `TestDonna` that asks the user for three names. The program prints the word `true` if the name Donna was one of the names entered by the user. Otherwise it prints the word `false`. Here is a sample run,

```
Name 1: Anna
Name 2: Lucille
Name 3: Donna

true
```

11. Write a Java program with the class name `SelectedNames` that asks the user for five names. The program prints only the names that begin with the letter `A`. For example,

```
Name 1: Anna
Name 2: Lucille
Name 3: Andrew
Name 4: Frederic
Name 5: James

Anna
Andrew
```

12. Using an ASCII table, put the following characters in ascending order:

```
b   5   !   =   7   ?   Y
```

13. Put the following strings of text in asceding order:

```
4   10   ban   2   1013   banana   Tomato   416-393-1622
```

14. True or false: The following code segment

```
char letter1 = 'A';
char letter2 = 'A';

if (letter1 == letter2) {
    Stdout.print("Letters are equal");
}
```

will print

```
Letters are equal
```

15. True or false: Two String variables can point to the same memory address.

4.3 Case Study: Encryption

Definitions
Cryptography: The science that studies hidden (“crypt”) writings (“graphy”).

Encryption is the process by which we transform a readable message, the *plaintext*, into an unreadable one, the *ciphertext*.

Decryption is the reverse process of encrypting, i.e., turning a ciphertext into plaintext. The algorithm used to perform an encryption/decryption is the *cipher*.

The two most common types of encryption are: *Symmetric* and *Asymmetric*.

In symmetric encryption, plaintext is transformed into ciphertext by means of a private key. A key is a password that is used by the cipher to perform the encryption. The same key is used when decrypting the ciphertext, applying the private key in a reverse process. In symmetric encryption the key is kept private for both processes. The person who encrypts a message has the private key, and the person who decrypts the message also has the private key. No one else has this key. Otherwise the ciphertext can be easily decrypted.

In asymetric encription, two different keys are used: One for encrypting and another for decrypting. The encryption key is made public, i.e., any sender can use it to encrypt their messages. The decryption key is kept private and is used for decrypting the ciphertext. Most encryption schemes today used by financial institutions, computer networks, goverment agencies, are asymmetric. Clients have a key that is published, while the institutions keep the private key. One of the best know and most widely implemented encryption schemes is RSA (Rivest-Shamir-Adleman, 1977) which determines its keys by the use of prime numbers. It is the infeasability of factoring large composite numbers into their prime factors that makes this algorithm particularly powerful. We will study it with more depth later.

Examples of Symmetric Encryption

Messages without a key or password

Messages can be enctryped by applying an operation to the characters of the plaintext without using any kind of key. Decryption consists of applying the inverse operation. For example, taking each character and mapping it to another character, like mapping a character to the next one in the alphabet or the ASCII table.

Attempt to decrypt the following messages of increasing difficulty

a) `UIF!DBU!BUF!VIF!SBU`

b) `UNEZZ=JR@SITSREZZ`

c) `XGHO7OZ2JT4CQ6JTK`

The cipher in the last case takes a character and it uses its position in the message as a way of encrypting.

Messages with a key or password

When encrypting using a key, we perform some kind of juxtaposition of the key and the plaintext to obtain the ciphertext. When using a key, the first position of the message is 0, not 1. Here are some examples:

a) In this example, the cipher consists of adding the ASCII values of the plaintext characters with the ASCII values of the key, repeating the key as necessary so as to encrypt the entire plaintext:

```
Ciphertext(i) = Plaintext(i) + key(i % key.length())
```

Plaintext: `We are going to the beach` Key: `iPhone`
Cipher: `iPhoneiPhoneiPhoneiPhonei`
Ciphertext: `Àµ?ÐàÊ?·×ØÜÌ?Ä×?âÍÎpÊÔÏÈÑ`

When applying this cipher, it could happen that the ASCII value obtained for a character is larger than 255. Because most programming languages use only one byte to store a character, it is a good idea to “wrap” around the result using the modulus operation to guarantee that the ending result is in the range [0, 255]. Therefore,

```
Ciphertext(i) = Ciphertext(i) % 256
```

Because of the use of both a key and an algorithm, the ciphertext starts becoming more random. It is through analysis of patterns that code breaking becomes more possible. The more random our ciphertext is, the harder it is to decipher a message.

b) Now we turn to a cipher that is more complex and much harder to break. In order to encrypt and decrypt successfully we need to make sure that the cipher uses a *one to one* function. An odd-degree polynomial is a good choice. For example

$$f(x) = nx^3$$

where *x* is the ASCII code of a plaintext character and *n* is the ASCII code of the key. Let's say that the plaintext is the word PHONE and the key is AMANDA. The ciphertext is:

plaintext	ASCII (x)	key	ASCII (n)	$f(x) = nx^3$	4 bytes (hex)
P	80	A	65	33280000	01 FB D0 00
H	72	M	77	28740096	01 B6 8A 00
O	79	A	65	32047535	01 E9 01 AF
N	78	N	78	37015056	02 34 CE 10
E	69	D	68	22338612	01 54 DC 34

The bytes in the last column is the final transmission. In order to decrypt the message we need both the key and the function.

4.3 Exercises

1. Write a Java program to encrypt a message using the following cipher:
   a. Each character of the plaintext is changed to the next ASCII value
   b. If a plaintext character has an ASCII value of 255, its ciphertext character becomes ASCII 0 (note: position starts at 1 not 0)[2]

   Test your program with the following

plaintext	ciphertext
`We are having lunch`	`Xf!bsf!ibwjoh!mvodi`
`It's a rainy day`	`Ju(t!b!sbjoz!ebz`
`Name: Wally`	`Obnf;!Xbmmz`

2. Write a Java program to encrypt a message using the following cipher:
   a. If a plaintext character is in an odd position, it becomes its ASCII + 1; if a plaintext character is in an even position, it becomes its ASCII - 1
   b. If a plaintext character has ASCII value 255 and is in an odd position, its ciphertext ASCII becomes 0; if a plaintext character has ASCII value 0 and is in an even position, its ciphertext ASCII becomes 255. (note: position starts at 1 not 0)

   Test your program with the following:

plaintext	ciphertext
`Small world`	`Tlbkmxnske`
`Price: $23.75`	`Qqjbf9!#32/66`
`416-967-1111`	`507,:58,2020`

[2] We are using the modulus operator % with the position. Therefore, we cannot use zero.

3. Write a Java program to encrypt a message using the following cipher: Each plaintext character is encrypted by adding to its ASCII value its position in the message mod 10. For example, the word `DOG` is encrypted to `EQJ` by: (note: position starts at 1 not 0)

PLAINTEXT	ASCII	POSITION mod 10	Clpher	ClpherTEXT
D	68	1 % 10 = 1	68 + 1	E
O	79	2 % 10 = 2	79 + 2	Q
G	71	3 % 10 = 3	71 + 3	J

Test your program with the following:

plaintext	ciphertext
`HOLIDAYS SOON`	`IQOMIG`[)SPQQ`
`She likes to paint with oils`	`Tjh$qorm| uq#tfou|)wjvk$tos{`
`(3 + 4) - 23 / 11`	`)5#/%:0(6 35#3%78`

4. Write a Java program that decrypts messages that have been encrypted with the cipher and the key of page 21, part b).

   a. In preparation for this exercise, write first a program that takes the string `Hello how are you today?` and prints each word of the string in a separate line. The output of the program is:

```
Hello
how
are
you
today?
```

   Once you have written this small program, it will help in understanding how to process the cypher text given below.

   b. Now proceed to write a program to decipher the ciphertext below using the cipher and the key of page 21, part b). In your program you

will need to perform the cubed root at some point. Make sure to round the result of the cubed with the method `Math.round()`. This happens because precision in the computer is finite. The method returns a `long`. It may be necessary to cast the result to an integer if the variables you use are integers.

Then, using your program decrypt the following message:

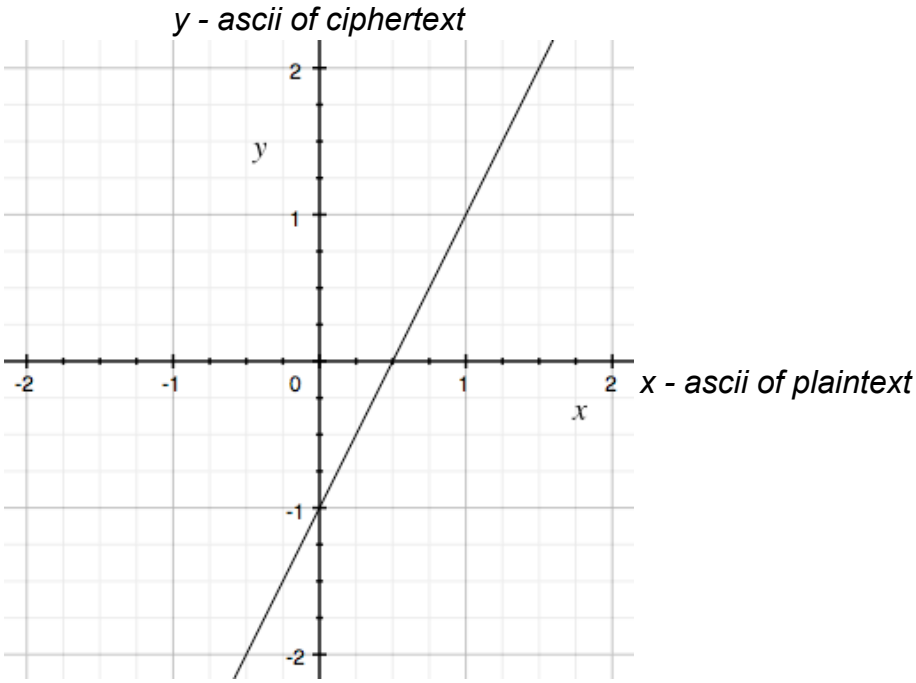
```
24261120 105307587 81881280 90294750 68000000 59323745

115151465 2523136 98856875 106675218 92998908 86515000 2335905
```

5. The following ciphertext has been intercepted:

```
133 221 221 213 209 201 63 153 221 219 229 231 201 227
```

and we know that each number can be deciphered to a plaintext character. Each plaintext character was encrypted using its ASCII code, and the following cipher was applied:



Write a Java program that decrypts the message.

6. The plaintext

```
PIZZA PLEASE
```

is being encrypted with a password in the cipher. The ASCII code of each plaintext character is being added to the ASCII code of the password and 30 is being subtracted. The resulting ciphertext is

```
smvmdDl_heoX
```

What is the password? Show your calculations.

7. Write a Java program that encrypts the plaintext of the previous question using the password that you determined. The ciphertext should be the same as the ciphertext of the previous question.

8. Let *x* be the alphabet position for the plaintext characters. The resulting values of the ciphertext characters can be indefinitely large. Which of the following can be used as ciphers? Why or why not?

a) $f(x) = -2x^3 + 5x - 1$ b) $f(x) = 3(x-32)^2, 0 \le x$ c) $f(x) = \sqrt{x}, 0 \le x$

d) $f(x) = \sin(x)$ e) $f(x) = \begin{cases} -2x+1, 0 \le x \le 96 \\ x^2, x \ge 97 \end{cases}$ f) $f(x) = 2^x - x$

9. Write a Java program that encrypts a plaintext according to the following cipher, where *x* is the ASCII value of each character:

$$f(x) = 2x - 30$$

Display the ciphertext as numbers, that is, do not convert the ciphertext numbers back to characters. Test your program with the following plaintexts:

plaintext	ciphertext
`My computer`	`124 212 34 168 192 188 194 204 202 172 198`
`Soft drink`	`136 192 174 202 34 170 198 180 190 184`
`Ernie`	`108 198 190 180 172`

10. Write a Java program that takes the ciphertext of the previous question and encrypts it again according to the cipher

$$f(x) = -x + 5$$

For example, if we use the ciphertext `108 198 190 180 172`, the program encrypts by applying the cipher to each one of the number. The result will be `-103 -193 -185  -175 -167`

This means that the original plaintext `Ernie` is encrypted (actually doubly encrypted) to these numbers. When you run your program you can do each number separately. The program does not need to handle all ciphers at the same time.

11. The following exercise shows that a double encryption can be done with a single mathematical function. This is called function composition and here we compose the ciphers of exercises 9 and 10. It looks a bit complicated, like this:

$$f(2x-30) = -x+5 = -(2x-30)+5 = -2x+30+5 = -2x+35$$
$$f(x) = -2x+35$$

Write a Java program that encrypts using the cipher $f(x) = -2x+35$. This will encrypt the original message `Ernie` into the codes `-103 -193 -185 -175 -167` in a single encryption.

12. Write a Java program that declares two string variables. The first string variable is called `labels` and contains the letters

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

The second string variable is called `rotor1` and contains the letters

```
OKDEMFZGXHCTNPQRASLUVWJBYI
```

The program asks the user for a letter, for example `B`. The program locates the position of `B` in `labels`. The position in this case is 1. Then the program outputs the corresponding letter in position 1 in the variable `rotor1`: In this case `K`. By doing this, we have encrypted the letter `B` into the letter `K`

13. Write a Java program that does the reverse of question 12. For example, if the user types the letter `K` the program prints out the letter `B`. This program decrypts what we encrypt with the program of question 12.

14. Now create another variable called `rotor2` that contains the letters

`KDMIOZGXHESFCTNPAQUVWYRLJB`

and add it to the program of question 12. Modify the program so that when the user enters a plaintext letter, the following happens:

a. Create a variable called `letter` and ask the user for a letter to encrypt
b. Find `letter` in `labels` and store its location in an integer variable `i`
c. Look now in the variable `rotor1` and store in `letter` the letter that is in position `i`
d. Find `letter` in `labels` and store its location in an integer variable `i`
e. Look now in the variable `rotor2` and store in `letter` the letter that is in position `i`
f. Print the variable `letter`

This program encrypts a letter by going through two levels of encryption. We have created a program that encrypts the way that Enigma encrypts with its first 2 rotors. Enigma has 3 rotors so we will write the remaining code to encrypt with all 3 of them.

15. The previous two questions contain the ciphers for rotor 1 and rotor 2. We now add rotor 3 and we have:

- rotor1 is `OKDEMFZGXHCTNPQRASLUVWJBYI`
- rotor2 is `KDMIOZGXHESFCTNPAQUVWYRLJB`
- rotor3 is `ZABIUKRXTPJFCHNEDQOWYLGVSM`

Using the three rotors in your program, confirm these encryptions

plaintext	ciphertext
TONIGHTISPARTYTIME	GAQNKFBOHMVKTGJZPW
IAMCOMINGHOME	FNLFUSQGPEJUN
WELOVEMUSIC	UIVILXMKHYO

16. Using the encryption rotors of question 15, a message was encrypted. The message is,

    `WNGFJVZPLL`

    Make a copy of the encryption program to a separate file and then modify it so that it will decrypt the message. The decrypted message answers the question “What do we all look forward to?”

17. Extend the rotors of question 15 so that the messages can contain spaces and numbers. For clarity, the spaces of the plaintext are denoted with dashes. Note that if a dash appears in the chiphertext, it does not necessarily mean that it’s a space.

- labels is `ABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789`
- rotor1 is `OK4DE17MF5ZGX-HC8TN0PQR3ASL2UV9W6JBYI`
- rotor2 is `KDMIOZ09GXHE7S1FCT65N-PA3Q8UVW2YRLJB4`
- rotor3 is `ZAB-IUKR0XT6PJFCH2N9EDQ5O1W37YLGV4SM8`

With these new rotors, confirm these encryptions

plaintext	ciphertext
`MARCH-1-2017`	`ZYS3AX96SG5F`
`ALAN-TURING`	`77LA016B4R9`
`GOOD-FOOD`	`XZA60WUKH`

(Note the difference between zero `0` and the letter “O” `O`)