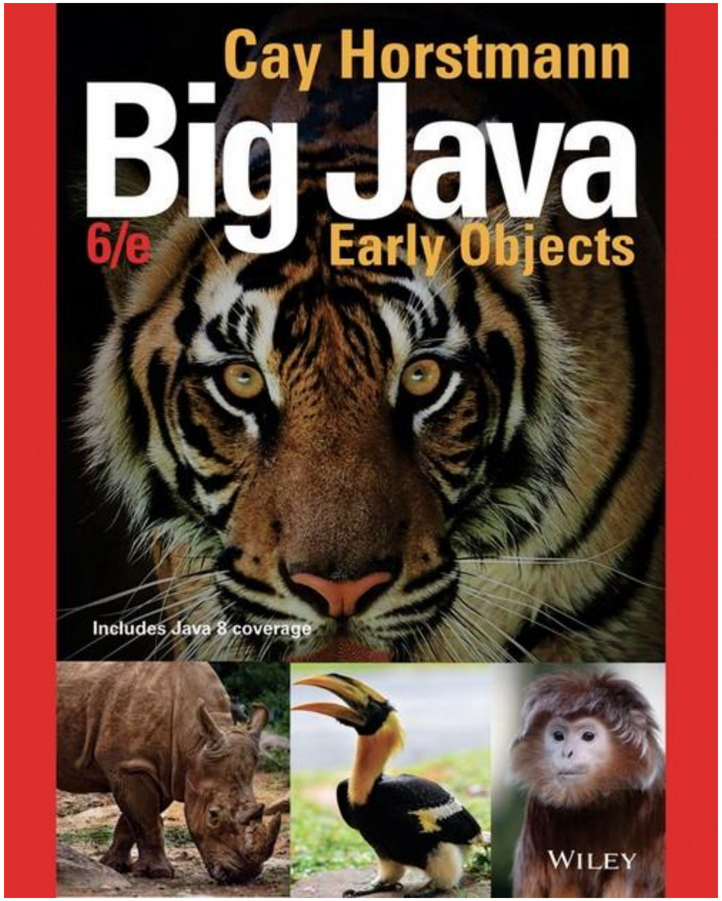


Chapter 11 – Input/Output & Exception Handling Part II

Text Input and Output

- The `next` method of the `Scanner` class reads a string that is delimited by white space.
- A loop for processing a file

```
while (in.hasNext())
{
    String input = in.next();
    System.out.println(input);
}
```

- If the input is `"Mary had a little lamb"`, the loop prints each word on a separate line

```
Mary
Had
A
Little
lamb
```

Text Input and Output

- The `next` method returns any sequence of characters that is not white space.
- **White space** includes: spaces, tab characters, and the newline characters that separate lines.
- These strings are considered “words” by the next method

```
Snow.
1729
C++
```

Text Input and Output

- When `next` is called:
  - Input characters that are white space are consumed - removed from the input
  - They do not become part of the word
  - The first character that is **not** white space becomes the first character of the word
  - More characters are added until
    - Either another white space character occurs
    - Or the end of the input file has been reached
- If the end of the input file is reached before any character was added to the word
  - a “no such element exception” occurs.

Text Input and Output

- To read just words and discard anything that isn't a letter:
    - Call useDelimiter method of the Scanner class

```
Scanner in = new Scanner(. . .);
in.useDelimiter("[^A-Za-z]+");
. . .
```

- The word separator becomes any character that is **not** a letter.
- Punctuation and numbers are not included in the words returned by the next method.

Text Input and Output - Reading Characters

- To read one character at a time, set the delimiter pattern to the empty string:

```
Scanner in = new Scanner(. . .);
in.useDelimiter("");
```

- Now each call to `next` returns a string consisting of a single character.
- To process the characters:

```
while (in.hasNext())
{
   char ch = in.next().charAt(0);
   Process ch
}
```

Text Input and Output - Classifying Characters

The `Character` class has methods for classifying characters.

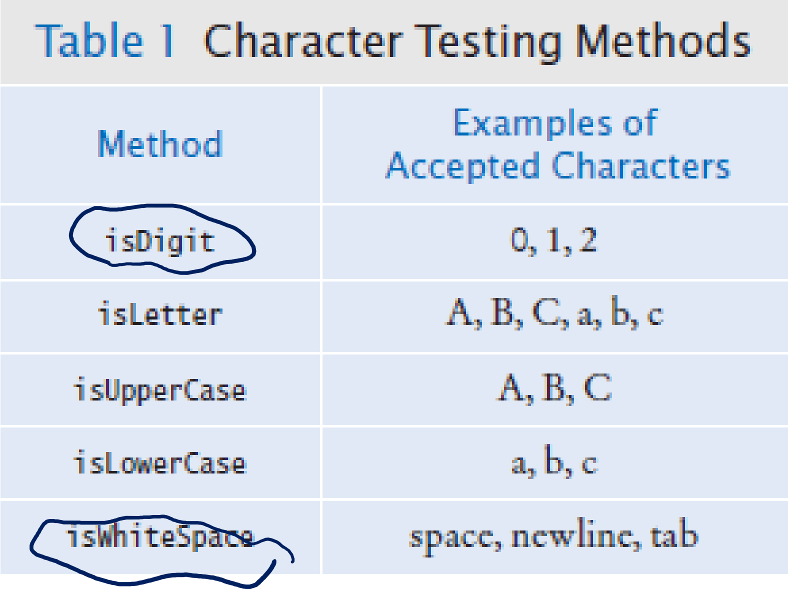


Table 1 Character Testing Methods

Method	Examples of Accepted Characters
isDigit	0, 1, 2
isLetter	A, B, C, a, b, c
isUpperCase	A, B, C
isLowerCase	a, b, c
isWhiteSpace	space, newline, tab

Text Input and Output - Reading Lines

- The `nextLine` method reads a line of input and consumes the newline character at the end of the line:

```
String line = in.nextLine();
```

- The `hasNextLine` method returns `true` if there are more input lines, `false` when all lines have been read.

- Example: process a file with population data from the CIA Fact Book with lines like this:

```
China 1330044605
India 1147995898
United States 303824646
...
```

Text Input and Output - Reading Lines

- Read each input line into a string

```
while (in.hasNextLine())
{
   String line = nextLine();
   Process line.
}
```

- Then use the `isDigit` and `isWhitespace` methods to find out where the name ends and the number starts.
- To locate the first digit:

```
int i = 0;
while (!Character.isDigit(line.charAt(i))) { i++; }
```

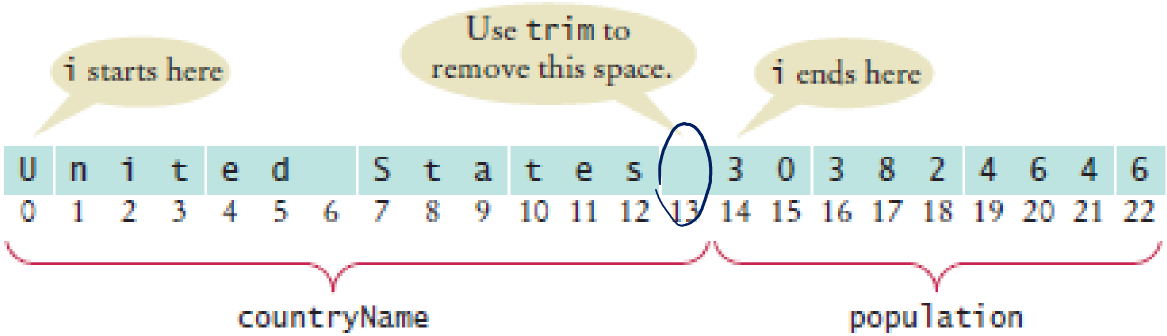
- To extract the country name and population:

```
String countryName = line.substring(0, i);
String population = line.substring(i);
```

Text Input and Output - Reading Lines

- Use `trim` to remove spaces at the beginning and end of string:

```
countryName = countryName.trim();
```



- Note that the population is stored in a string.

Text Input and Output - Scanning a String

- Occasionally easier to construct a new `Scanner` object to read the characters from a string:

```
Scanner lineScanner = new Scanner(line);
```

- Then you can use `lineScanner` like any other `Scanner` object, reading words and numbers:

```
String countryName = lineScanner.next();
while (!lineScanner.hasNextInt())
{
    countryName = countryName + " " +
   lineScanner.next();
}
int populationValue = lineScanner.nextInt();
```

Text Input and Output - Converting Strings to Numbers

- If a string contains the digits of a number.
  - Use the `Integer.parseInt` or `Double.parseDouble` method to obtain the number value.
- If the string contains `"303824646"`
  - Use `Integer.parseInt` method to get the integer value

```
int populationValue = Integer.parseInt(population);
// populationValue is the integer 303824646
```

- If the string contains `"3.95"`
  - Use `Double.parseDouble`

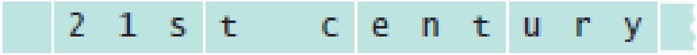
```
double price = Double.parseDouble(input);
// price is the floating-point number 3.95
```

- The string must not contain spaces or other non-digits. Use `trim`:

```
int populationValue = Integer.parseInt(population.trim());
```

Avoiding Errors When Reading Numbers

- If the input is not a properly formatted number when calling `nextInt` or `nextDouble` method



2 1 s t c e n t u r y

- For example, if the input contains characters:
  - White space is consumed and the word 21st is read.
  - 21st is not a properly formatted number
  - Causes an input mismatch exception in the `nextInt` method.
- If there is no input at all when you call `nextInt` or `nextDouble`,
  - A “no such element exception” occurs.
- To avoid exceptions, use the `hasNextInt` method

```
if (in.hasNextInt()) { int value = in.nextInt(); . . . }
```

Mixing Number, Word, and Line Input

- The `nextInt`, `nextDouble`, and `next` methods do **not** consume the white space that follows the number or word.
- This can be a problem if you alternate between calling `nextInt`/`nextDouble`/`next` and `nextLine`.
- Example: a file contains country names and populations in this format:

```
China
1330044605
India
1147995898
United States
303824646
```

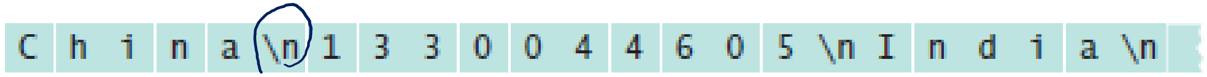
Mixing Number, Word, and Line Input

- The file is read with these instructions:

```
while (in.hasNextLine())
{
   String countryName = in.nextLine();
   int population = in.nextInt();
   Process the country name and population.
}
```

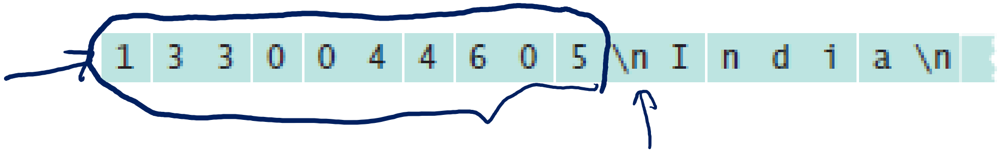
Mixing Number, Word, and Line Input

- Initial input



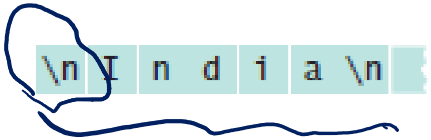
C h i n a \n 1 3 3 0 0 4 4 6 0 5 \n I n d i a \n

- Input after first call to `nextLine`



1 3 3 0 0 4 4 6 0 5 \n I n d i a \n

- Input after call to `nextInt`



\n I n d i a \n

  - `nextInt` did **not** consume the newline character

- The second call to `nextLine` reads an empty string!

- The remedy is to add a call to `nextLine` after reading the population value:

```
String countryName = in.nextLine();
int population = in.nextInt();
in.nextLine(); // Consume the newline
```

Formatting Output

- There are additional options for `printf` method.
- Format flags

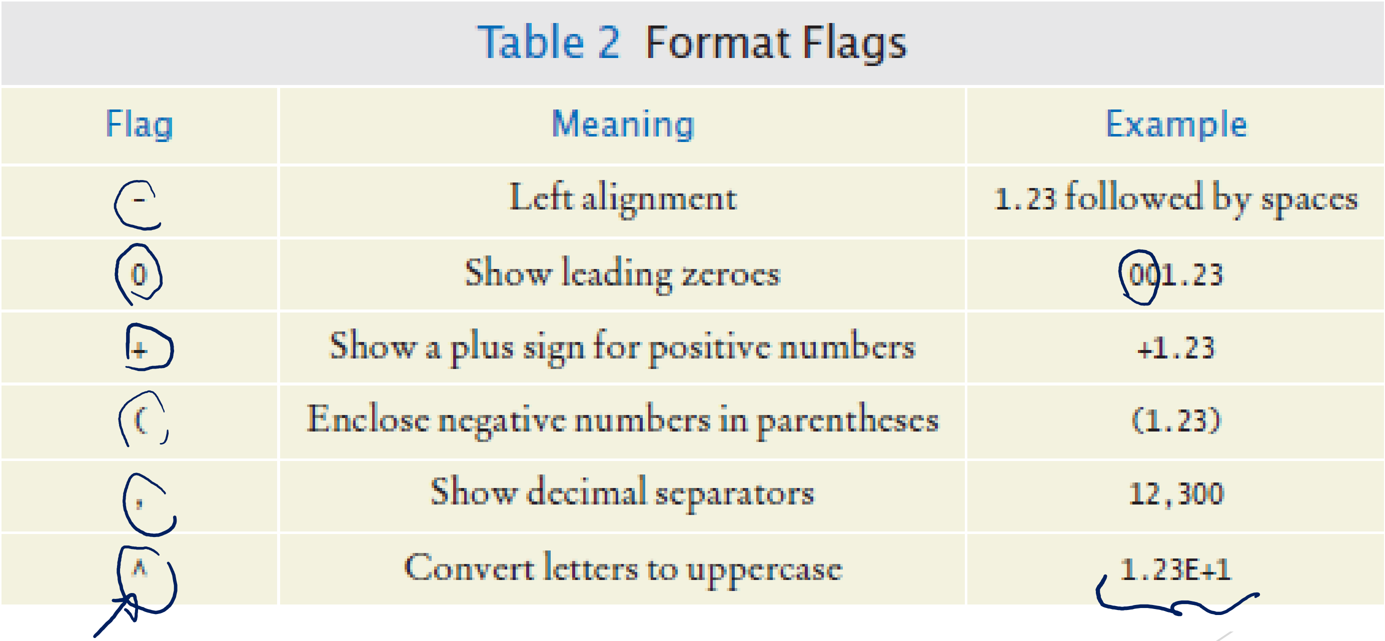


Table 2 Format Flags

Flag	Meaning	Example
-	Left alignment	1.23 followed by spaces
0	Show leading zeroes	001.23
+	Show a plus sign for positive numbers	+1.23
(	Enclose negative numbers in parentheses	(1.23)
,	Show decimal separators	12,300
^	Convert letters to uppercase	1.23E+1

Formatting Output

- Example: print a table of items and prices, each stored in an array

```
Cookies:        3.20
Linguine:       2.95
Clams:         17.29
```

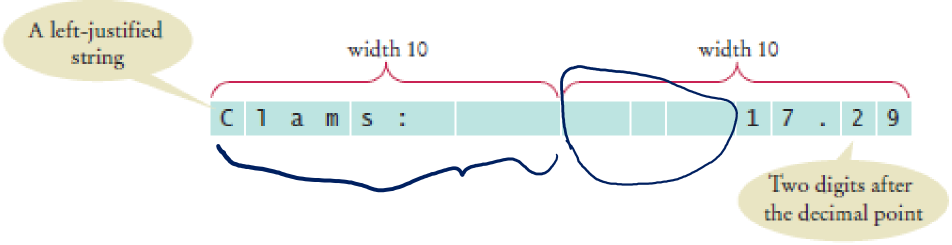
- The item strings line up to the left; the numbers line up to the right.

Formatting Output

- To specify left alignment, add a hyphen (-) before the field width:

```
System.out.printf("%-10s%10.2f", items[i] + ":",
  prices[i]);
```

- There are two format specifiers: `"%-10s%10.2f"`
- `%-10s`
  - Formats a left-justified string.
  - Padded with spaces so it becomes ten characters wide
- `%10.2f`
  - Formats a floating-point number
  - The field that is ten characters wide.
  - Spaces appear to the left and the value to the right
- The output



```
Clams:         17.29
```

Formatting Output

- A format specifier has the following structure:
  - The first character is a %.
  - Next are optional “flags” that modify the format, such as – to indicate left alignment.
  - Next is the field width, the total number of characters in the field (including the spaces used for padding), followed by an optional precision for floating-point numbers.
  - The format specifier ends with the format type, such as f for floating-point values or s for strings.

Formatting Output

- Format types

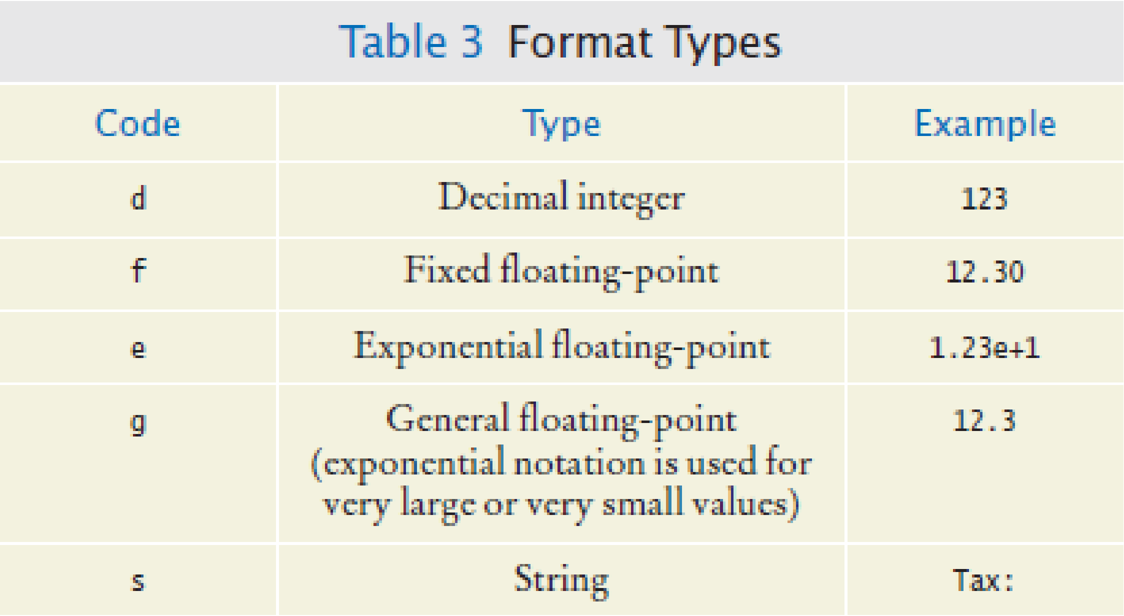


Table 3 Format Types

Code	Type	Example
d	Decimal integer	123
f	Fixed floating-point	12.30
e	Exponential floating-point	1.23e+1
g	General floating-point (exponential notation is used for very large or very small values)	12.3
s	String	Tax:

Self Check 11.6

Suppose the input contains the characters `Hello, World!`. What are the values of `word` and `input` after this code fragment?

```
String word = in.next();
String input = in.nextLine();
```

Answer: `word` is `"Hello"`, and `input` is `"World!"`

Self Check 11.7

Suppose the input contains the characters 995.0 Fred. What are the values of number and input after this code fragment?

```
int number = 0;
if (in.hasNextInt()) { number = in.nextInt(); }
String input = in.next();
```

Answer: Because 995.0 is not an integer, the call in.hasNextInt() returns false, and the call in.nextInt() is skipped. The value of number stays 0, and input is set to the string "995.0".

Self Check 11.8

Suppose the input contains the characters 6E6 6,995.00. What are the values x1 and x2 after this code fragment?

```
double x1 = in.nextDouble();
double x2 = in.nextDouble();
```

Answer: x1 is set to 6000000. Because a comma is not considered a part of a floating-point number in Java, the second call to nextDouble causes an input mismatch exception and x2 is not set.

Self Check 11.9

Your input file contains a sequence of numbers, but sometimes a value is not available and marked as N/A. How can you read the numbers and skip over the markers?

Answer: Read them as strings, and convert those strings to numbers that are not equal to N/A:

```
String input = in.next();
if (!input.equals("N/A"))
{
   double value = Double.parseDouble(input);
   Process value
}
```

Self Check 11.10

How can you remove spaces from the country name in Section 11.2.4 without using the `trim` method?

Answer: Locate the last character of the country name:

```
int j = i – 1;
while (!Character.isWhiteSpace(line.charAt(j)))
{
   j--;
}
```

Then extract the country name:

```
String countryName = line.substring(0, j + 1);
```

Command Line Arguments

- You can run a Java program by typing a command at the prompt in the command shell window
  - Called “invoking the program from the command line”
- With this method, you can add extra information for the program to use
  - Called **command line arguments**
- Example: start a program with a command line

```
java ProgramClass -v input.dat
```

- The program receives the strings `"-v"` and `"input.dat"` as command line arguments
- Useful for automating tasks

Command Line Arguments

- Your program receives its command line arguments in the `args` parameter of the main method:

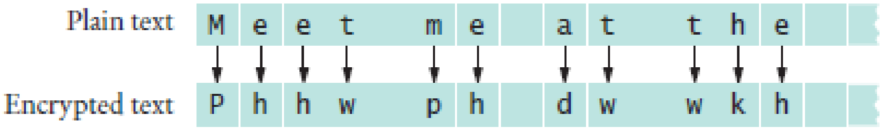
```
public static void main(String[] args)
```

- In the example, `args` is an array of length 2, containing the strings

```
args[0]: "-v”
args[1]: "input.dat"
```

Command Line Arguments

- Example: a program that encrypts a file
  - Use a Caesar Cipher that replaces A with a D, B with an E, and so on
  - Sample text



Plain text	M	e	e	t		m	e		a	t		t	h	e	
Encrypted text	P	h	h	w		p	h		d	w		w	k	h	

- The program will take command line arguments
  - An optional `-d` flag to indicate decryption instead of encryption
  - The input file name
  - The output file name
- To encrypt the file `input.txt` and place the result into `encrypt.txt`

```
java CaesarCipher input.txt encrypt.txt
```

- To decrypt the file `encrypt.txt` and place the result into `output.txt`

```
java CaesarCipher -d encrypt.txt output.txt
```

section_3/CaesarCipher.java

```java
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;

/**
    This program encrypts a file using the Caesar cipher.
*/
public class CaesarCipher
{
    public static void main(String[] args) throws FileNotFoundException
    {
        final int DEFAULT_KEY = 3;
        int key = DEFAULT_KEY;
        String inFile = "";
        String outFile = "";
        int files = 0; // Number of command line arguments that are files

```

Continued

section_3/CaesarCipher.java

```
      for (int i = 0; i < args.length; i++)
      {
         String arg = args[i];
         if (arg.charAt(0) == &apos;-&apos;)
         {
            // It is a command line option

            char option = arg.charAt(1);
            if (option == &apos;d&apos;) { key = -key; }
            else { usage(); return; }
         }
         else
         {
            // It is a file name

            files++;
            if (files == 1) { inFile = arg; }
            else if (files == 2) { outFile = arg; }
         }
      }
      if (files != 2) { usage(); return; }
```

Continued

section_3/CaesarCipher.java

```
       Scanner in = new Scanner(new File(inFile));
       in.useDelimiter(""); // Process individual characters
       PrintWriter out = new PrintWriter(outFile);

       while (in.hasNext())
       {
          char from = in.next().charAt(0);
          char to = encrypt(from, key);
          out.print(to);
       }
       in.close();
       out.close();
    }

```

Continued

section_3/CaesarCipher.java

```java
   /**
      Encrypts upper- and lowercase characters by shifting them
      according to a key.
      @param ch the letter to be encrypted
      @param key the encryption key
      @return the encrypted letter
   */
   public static char encrypt(char ch, int key)
   {
      int base = 0;
      if (&apos;A&apos; <= ch && ch <= &apos;Z&apos;) { base = &apos;A&apos;; }
      else if (&apos;a&apos; <= ch && ch <= &apos;z&apos;) { base = &apos;a&apos;; }
      else { return ch; } // Not a letter
      int offset = ch - base + key;
      final int LETTERS = 26; // Number of letters in the Roman alphabet
      if (offset > LETTERS) { offset = offset - LETTERS; }
      else if (offset < 0) { offset = offset + LETTERS; }
      return (char) (base + offset);
   }

   /**
      Prints a message describing proper usage.
   */
   public static void usage()
   {
      System.out.println("Usage: java CaesarCipher [-d] infile outfile");
   }
}
```

Self Check 11.11

If the program is invoked with

```
java CaesarCipher -d file1.txt
```

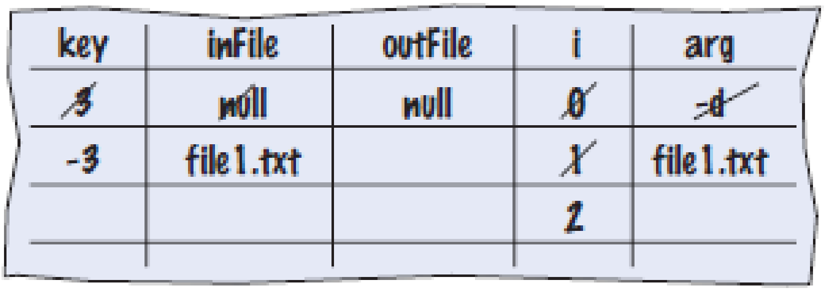
what are the elements of `args`?

Answer: `args[0]` is `"-d"` and `args[1]` is `"file1.txt"`

Self Check 11.12

Trace the program when it is invoked as in Self Check 11.

Answer:



key	inFile	outFile	i	arg
~~3~~	~~null~~	null	~~0~~	~~-d~~
-3	file1.txt		~~1~~	file1.txt
			2	

Then the program prints a message

```
Usage: java CaesarCipher [-d] infile outfile
```

Self Check 11.13

Will the program run correctly if the program is invoked with

```
java CaesarCipher file1.txt file2.txt –d
```

If so, why? If not, why not?

Answer: The program will run correctly. The loop that parses the options does not depend on the positions in which the options appear.

Self Check 11.14

Encrypt CAESAR using the Caesar cipher.

Answer: FDHVDU

Self Check 11.15

How can you modify the program so that the user can specify an encryption key other than 3 with a `-k` option, for example

```
java CaesarCipher -k15 input.txt output.txt
```

Answer: Add the lines

```
else if (option == 'k’)
{
   key = Integer.parseInt( args[i].substring(2));
}
```

after line 27 and update the usage information.