

Characters and Strings



Objectives

- After you have read and studied this chapter, you should be able to
 - Declare and manipulate data of the char data type.
 - Write string processing program, applicable in areas such as bioinformatics, using String, StringBuilder, and StringBuffer objects.
 - Differentiate the three string classes and use the correct class for a given task.
 - Specify regular expressions for searching a pattern in a string.
 - Use the Pattern and Matcher classes.
 - Compare the String objects correctly.



Characters

- In Java, single characters are represented using the data type **char**.
- Character constants are written as symbols enclosed in single quotes.
- Characters are stored in a computer memory using some form of encoding.
- *ASCII*, which stands for *American Standard Code for Information Interchange*, is one of the document coding schemes widely used today.
- Java uses **Unicode**, which includes ASCII, for representing **char** constants.



ASCII Encoding

	0	1	2	3	4	5	6	7	8	9
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
10	lf	vt	ff	cr	so	si	dle	dcl	dc2	dc3
20	cd4	nak	syn	etb	can	em	sub	esc	fs	gs
30	rs	us	sp	!	"	#	\$	%	&	'
40	()	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[\]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	}		~	del		

For example,
character 'O' is
79 (row value
70 + col value 9
= 79).



Unicode Encoding

- The *Unicode Worldwide Character Standard* (*Unicode*) supports the interchange, processing, and display of the written texts of diverse languages.
- Java uses the Unicode standard for representing **char** constants.

```
char ch1 = 'X';
```

```
System.out.println(ch1);
```

```
System.out.println( (int) ch1);
```

→ X
→ 88



Character Processing

```
char ch1, ch2 = 'X';
```

Declaration and
initialization

```
System.out.print("ASCII code of character X is " +  
                (int) 'X' );  
  
System.out.print("Character with ASCII code 88 is "  
                + (char)88 );
```

Type conversion between
int and char.

```
'A' < 'c'
```

This comparison returns
true because ASCII value
of 'A' is 65 while that of 'c'
is 99.



Strings

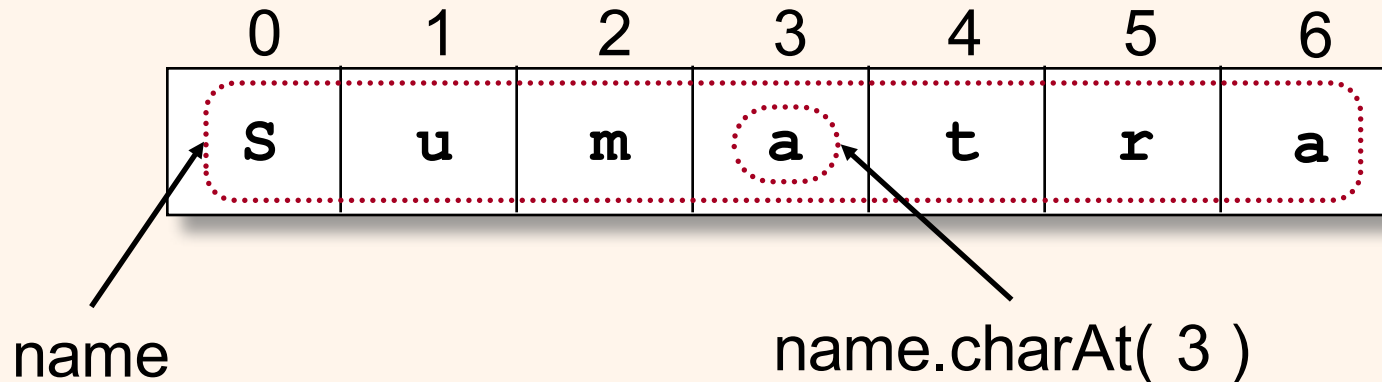
- A *string* is a sequence of characters that is treated as a single value.
- Instances of the **String** class are used to represent strings in Java.
- We can access individual characters of a string by calling the **charAt** method of the **String** object.



Accessing Individual Elements

- Individual characters in a String accessed with the `charAt` method.

```
String name = "Sumatra";
```



This variable refers to the whole string.

The method returns the character at position # 3.



Example: Counting Vowels

```
char    letter;
System.out.println("Your name:");
String  name = scanner.next(); //assume 'scanner' is created properly
int     numberOfCharacters = name.length();
int     vowelCount = 0;

for (int i = 0; i < numberOfCharacters; i++) {
    letter = name.charAt(i);

    if (    letter == 'a' || letter == 'A' ||
          letter == 'e' || letter == 'E' ||
          letter == 'i' || letter == 'I' ||
          letter == 'o' || letter == 'O' ||
          letter == 'u' || letter == 'U'
        ) {
        vowelCount++;
    }
}

System.out.print(name + ", your name has " + vowelCount + " vowels");
```

Here's the code to count the number of vowels in the input string.



Example: Counting 'Java'

```
int      javaCount      = 0;
boolean  repeat          = true;
String   word;
Scanner  scanner = new Scanner(System.in);
```

Continue reading words and count how many times the word Java occurs in the input, ignoring the case.

```
while ( repeat ) {
    System.out.print("Next word:");
    word = scanner.next();
```

```
    if ( word.equals("STOP") ) {
        repeat = false;
```

Notice how the comparison is done. We are not using the == operator.

```
    } else if ( word.equalsIgnoreCase("Java") ) {
        javaCount++;
    }
}
```



Other Useful String Operators

Method	Meaning
compareTo	Compares the two strings. <code>str1.compareTo(str2)</code>
substring	Extracts the a substring from a string. <code>str1.substring(1, 4)</code>
trim	Removes the leading and trailing spaces. <code>str1.trim()</code>
valueOf	Converts a given primitive data value to a string. <code>String.valueOf(123.4565)</code>
startsWith	Returns true if a string starts with a specified prefix string. <code>str1.startsWith(str2)</code>
endsWith	Returns true if a string ends with a specified suffix string. <code>str1.endsWith(str2)</code>

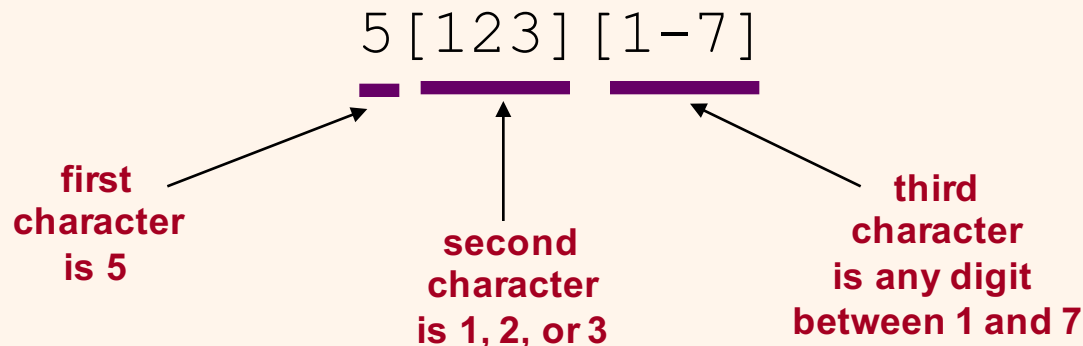
- See the String class documentation for details.



Pattern Example

- Suppose students are assigned a three-digit code:
 - The first digit represents the major (5 indicates computer science);
 - The second digit represents either in-state (1), out-of-state (2), or foreign (3);
 - The third digit indicates campus housing:
 - On-campus dorms are numbered 1-7.
 - Students living off-campus are represented by the digit 8.

The 3-digit pattern to represent computer science majors living on-campus is





Regular Expressions

- The pattern is called a *regular expression*.
- Rules
 - The brackets **[]** represent choices
 - The asterisk symbol ***** means zero or more occurrences.
 - The plus symbol **+** means one or more occurrences.
 - The hat symbol **^** means negation.
 - The hyphen **–** means ranges.
 - The parentheses **()** and the vertical bar **|** mean a range of choices for multiple characters.



Regular Expression Examples

Expression	Description
<code>[013]</code>	A single digit 0, 1, or 3.
<code>[0-9][0-9]</code>	Any two-digit number from 00 to 99.
<code>[0-9&&[^4567]]</code>	A single digit that is 0, 1, 2, 3, 8, or 9.
<code>[a-z0-9]</code>	A single character that is either a lowercase letter or a digit.
<code>[a-zA-z][a-zA-Z0-9_]*</code>	A valid Java identifier consisting of alphanumeric characters, underscores, and dollar signs, with the first character being an alphabet.
<code>[wb](ad eed)</code>	Matches <code>wad</code> , <code>weed</code> , <code>bad</code> , and <code>beed</code> .
<code>(AZ CA CO)[0-9][0-9]</code>	Matches <code>AZxx</code> , <code>CAxx</code> , and <code>COxx</code> , where <code>x</code> is a single digit.



The replaceAll Method

- The **replaceAll** method replaces all occurrences of a substring that matches a given regular expression with a given replacement string.

Replace all vowels with the symbol @

```
String originalText, modifiedText;  
  
originalText = ...;    //assign string  
  
modifiedText =  
    originalText.replaceAll("[aeiou]", "@");
```



The Pattern and Matcher Classes

- The `matches` and `replaceAll` methods of the `String` class are shorthand for using the `Pattern` and `Matcher` classes from the `java.util.regex` package.
- If `str` and `regex` are `String` objects, then

```
str.matches(regex);
```

is equivalent to

```
Pattern pattern = Pattern.compile(regex);  
Matcher matcher = pattern.matcher(str);  
matcher.matches();
```




The compile Method

- The **compile** method of the Pattern class converts the stated regular expression to an internal format to carry out the pattern-matching operation.
- This conversion is carried out every time the **matches** method of the String class is executed, so it is more efficient to use the compile method when we search for the same pattern multiple times.
- See the sample programs Ch9MatchJavaIdentifier2 and Ch9PMCountJava



The find Method

- The **find** method is another powerful method of the **Matcher** class.
 - It searches for the next sequence in a string that matches the pattern, and returns true if the pattern is found.
- When a matcher finds a matching sequence of characters, we can query the location of the sequence by using the **start** and **end** methods.
- See Ch9PMCountJava2



The String Class is Immutable

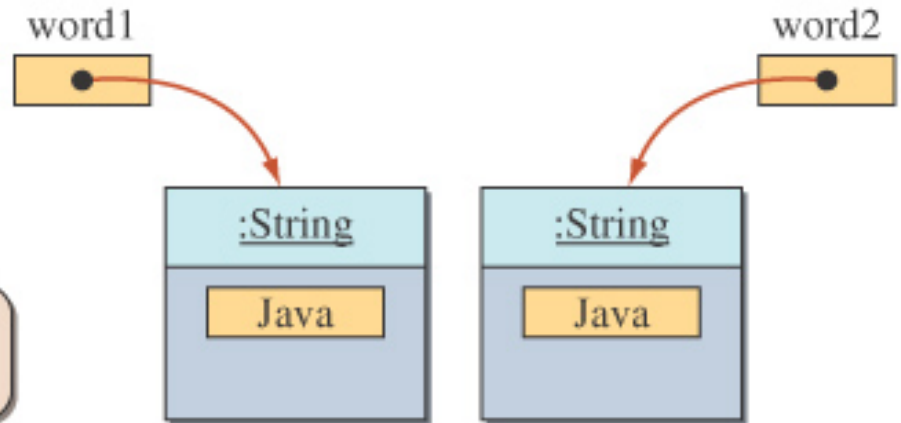
- In Java a String object is immutable
 - This means once a String object is created, it cannot be changed, such as replacing a character with another character or removing a character
 - The String methods we have used so far do not change the original string. They created a new string from the original. For example, substring creates a new string from a given string.
- The String class is defined in this manner for efficiency reason.



Effect of Immutability

```
String word1, word2;  
word1 = new String("Java");  
word2 = new String("Java");
```

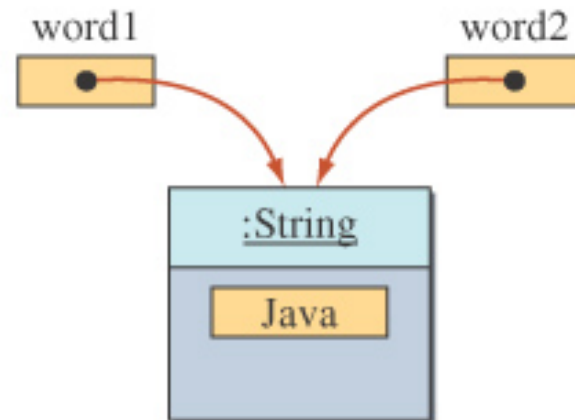
Whenever the **new** operator is used, there will be a new object.



```
String word1, word2;  
word1 = "Java";  
word2 = "Java";
```

We can do this because String objects are immutable.

Literal string constant such as "Java" will always refer to the one object.





The StringBuffer Class

- In many string processing applications, we would like to change the contents of a string. In other words, we want it to be mutable.
- Manipulating the content of a string, such as replacing a character, appending a string with another string, deleting a portion of a string, and so on, may be accomplished by using the **StringBuffer** class.

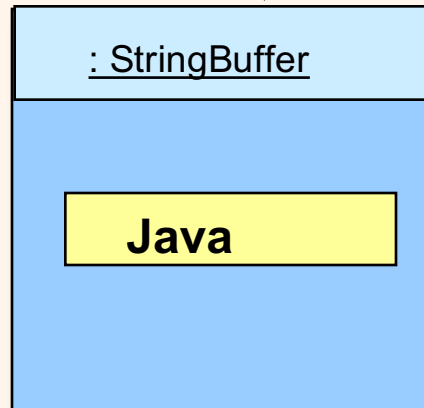
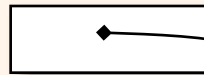


StringBuffer Example

```
StringBuffer word = new StringBuffer("Java");  
word.setCharAt(0, 'D');  
word.setCharAt(1, 'i');
```

Changing a string
Java to Diva

word

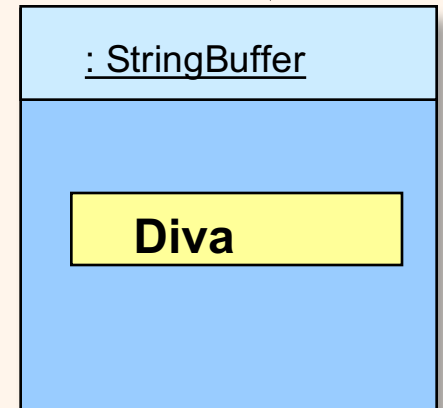
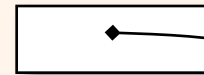


Before

```
word.setCharAt(0, 'D');  
word.setCharAt(1, 'i');
```



word



After



Sample Processing

Replace all vowels in the sentence with 'X'.

```
char        letter;
String      inSentence    = JOptionPane.showInputDialog(null, "Sentence:");
StringBuffer tempStringBuffer = new StringBuffer(inSentence);
int         numberOfCharacters = tempStringBuffer.length();

for (int index = 0; index < numberOfCharacters; index++) {

    letter = tempStringBuffer.charAt(index);

    if (letter == 'a' || letter == 'A' || letter == 'e' || letter == 'E' ||
        letter == 'i' || letter == 'I' || letter == 'o' || letter == 'O' ||
        letter == 'u' || letter == 'U' ) {
        tempStringBuffer.setCharAt(index, 'X');
    }
}

JOptionPane.showMessageDialog(null, tempStringBuffer );
```



The append and insert Methods

- We use the **append** method to append a String or StringBuffer object to the end of a StringBuffer object.
 - The method can also take an argument of the primitive data type.
 - Any primitive data type argument is converted to a string before it is appended to a StringBuffer object.
- We can insert a string at a specified position by using the **insert** method.



The StringBuilder Class

- This class is new to Java 5.0 (SDK 1.5)
- The class is added to the newest version of Java to improve the performance of the StringBuffer class.
- StringBuffer and StringBuilder support exactly the same set of methods, so they are interchangeable.
- There are advanced cases where we must use StringBuffer, but all sample applications in the book, StringBuilder can be used.
- Since the performance is not our main concern and that the StringBuffer class is usable for all versions of Java, we will use StringBuffer only in this book.



Bioinformatics

- Bioinformatics is a field of study that explores the use of computational techniques in solving biological problems.
- Genes are made of DNA (deoxyribonucleic acid), which is a sequence of molecules called *nucleotides* or *bases*.
- DNA provides instructions to the cell, so it serves a role similar to a computer program.
 - A cell is a computer that produces proteins (output) by reading instructions in DNA (program).
- The genetic information in DNA is encoded as a sequence of four chemical bases—adenine (A), guanine (G), cytosine (C), and thymine (T).



String Processing and Bioinformatics

- DNA is encoded as a sequence of bases.
- This information can be represented as a string of four letters—A, T, G, and C.
- Common operations biologists perform on DNA sequences can be implemented as string processing programs.
- See the sample programs
 - Ch9GCCContentt
 - Ch9TranscribeDNA
 - Ch9ReverseDNA



Problem Statement

Write an application that will build a word concordance of a document. The output from the application is an alphabetical list of all words in the given document and the number of times they occur in the document. The documents are a text file (contents of the file are an ASCII characters) and the output of the program is saved as an ASCII file also.



Overall Plan

- Tasks expressed in pseudocode:

```
while ( the user wants to process  
another file ) {
```

```
    Task 1: read the file;
```

```
    Task 2: build the word list;
```

```
    Task 3: save the word list to a file;
```

```
}
```

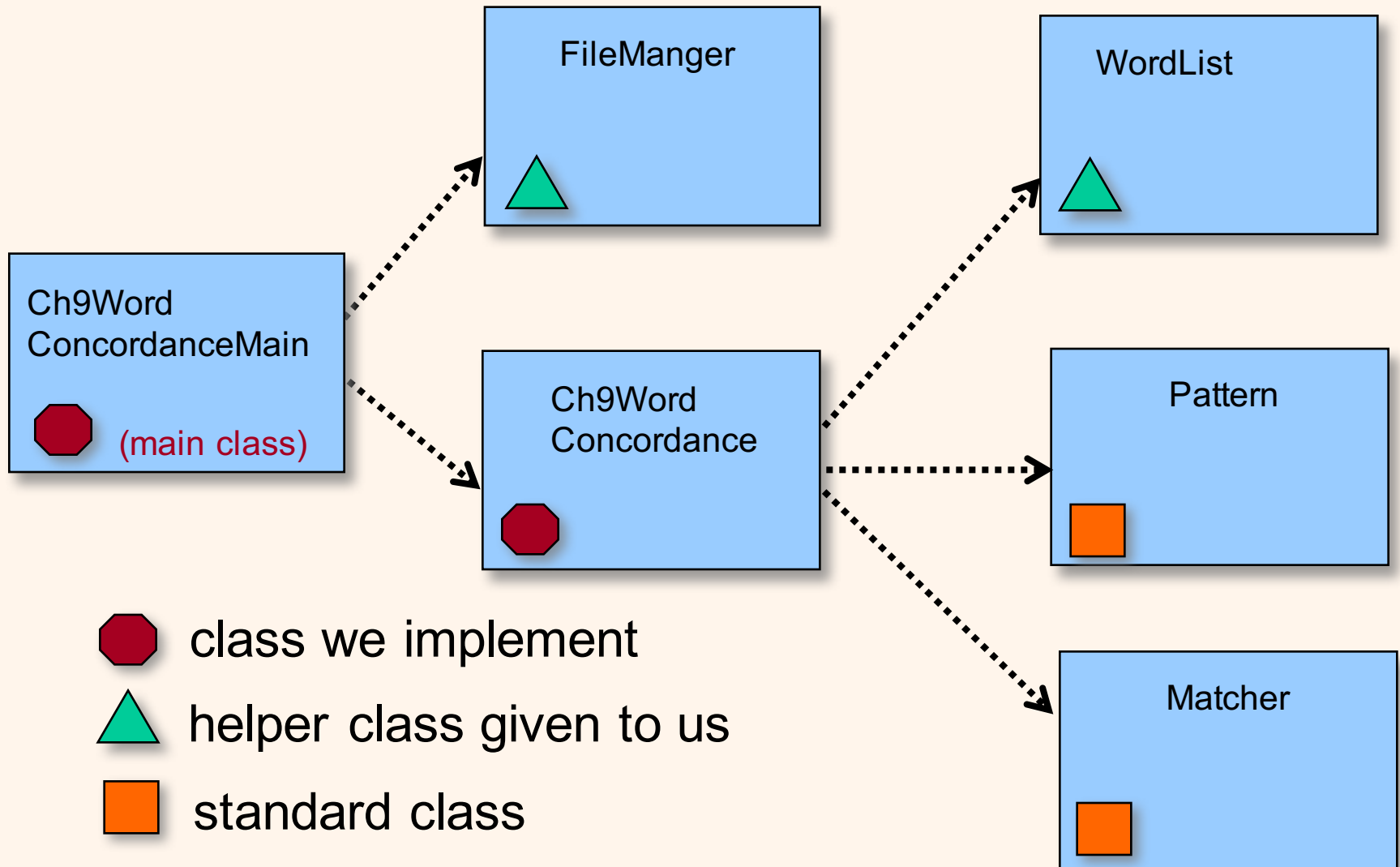


Design Document

Class	Purpose
Ch9WordConcordanceMain	The instantiable main class of the program that implements the top-level program control.
Ch9WordConcordance	The key class of the program. An instance of this class managers other objects to build the word list.
FileManager	A helper class for opening a file and saving the result to a file. Details of this class can be found in Chapter 12.
WordList	Another helper class for maintaining a word list. Details of this class can be found in Chapter 10.
Pattern/Matcher	Classes for pattern matching operations.



Class Relationships





Development Steps

- We will develop this program in four steps:
 1. Start with a program skeleton. Define the main class with data members. Begin with a rudimentary Ch9WordConcordance class.
 2. Add code to open a file and save the result. Extend the existing classes as necessary.
 3. Complete the implementation of the Ch9WordConcordance class.
 4. Finalize the code by removing temporary statements and tying up loose ends.



Step 1 Design

- Define the skeleton main class
- Define the skeleton
Ch9WordConcordance class that has
only an empty zero-argument constructor



Step 1 Code

Program source file is too big to list here. From now on, we ask you to view the source files using your Java IDE.

Directory: Chapter9/Step1

Source Files: Ch9WordConcordanceMain.java
Ch9WordConcordance.java



Step 1 Test

- The purpose of Step 1 testing is to verify that the constructor is executed correctly and the repetition control in the **start** method works as expected.



Step 2 Design

- Design and implement the code to open and save a file
- The actual tasks are done by the **FileManager** class, so our objective in this step is to find out the correct usage of the **FileManager** helper class.
- The **FileManager** class has two key methods: **openFile** and **saveFile**.



Step 2 Code

Directory: Chapter9/Step2

Source Files: Ch9WordConcordanceMain.java
Ch9WordConcordance.java



Step 2 Test

- The Step2 directory contains several sample input files. We will open them and verify the file contents are read correctly by checking the temporary echo print output to System.out.
- To verify the output routine, we save to the output (the temporary output created by the build method of Ch9WordConcordance) and verify its content.
- Since the output is a textfile, we can use any word processor or text editor to view its contents.



Step 3 Design

- Complete the **build** method of Ch9WordConcordance class.
- We will use the second helper class **WordList** here, so we need to find out the details of this helper class.
- The key method of the **WordList** class is the **add** method that inserts a given word into a word list.



Step 3 Code

Directory: Chapter9/Step3

Source Files: Ch9WordConcordanceMain.java
Ch9WordConcordance.java



Step 3 Test

- We run the program against varying types of input textfiles.
 - We can use a long document such as the term paper for the last term's economy class (don't forget to save it as a textfile before testing).
 - We should also use some specially created files for testing purposes. One file may contain one word repeated 7 times, for example. Another file may contain no words at all.



Step 4: Finalize

- Possible Extensions

- One is an integrated user interface where the end user can view both the input document files and the output word list files.
- Another is the generation of different types of list. In the sample development, we count the number of occurrences of each word. Instead, we can generate a list of positions where each word appears in the document.