COMP 401

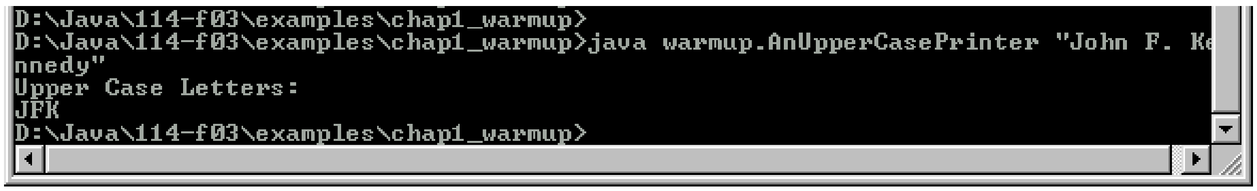
Prasun Dewan[1]

18. Iterator, Scanning & Streams

Now that we understand some basic principles of creating and using objects, let us revisit the scanning problem we saw earlier and redo it using objects. In this process, we will learn about an important kind of interface, called an enumeration or iterator interface, which applies to scanning and other kinds of problems. We will motivate this interface using another scanning problem – we will show that the solutions to the two problems can reuse code if such an interface is defined.

Previous Scanning Problem

Recall the scanning problem we implemented in the introduction.



```
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>java warmup.AnUpperCasePrinter "John F. Ke
nnedy"
Upper Case Letters:
JFK
D:\Java\114-f03\examples\chap1_warmup>
```

Figure 1 Finding Upper Case Letters

The following is the code we wrote:

```
package warmup;
public class AnUpperCasePrinter {
    public static void main(String[] args){
        if (args.length != 1) {
                System.out.println("Illegal number of arguments:" + args.length + ". Terminating 
program.");
                System.exit(-1);
        }
        System.out.println("Upper Case Letters:");
```

[1] © Copyright Prasun Dewan, 2009.

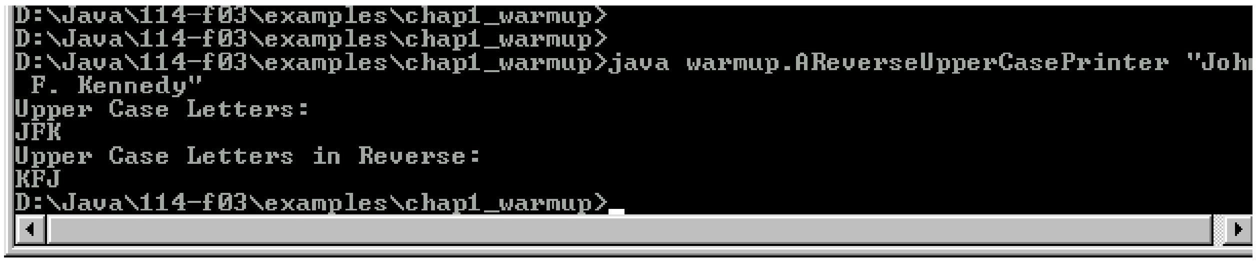
```
            int index = 0;
            while (index < args[0].length()) {
                        if (Character.isUpperCase(args[0].charAt(index)))
                                    System.out.print(args[0].charAt(index));
                                    index++;
                        }
            }
            System.out.println();
    }

}
```

Figure 2 An UpperCasePrinter

Scanning Problem Extension

Consider an extension of the problem above, in which we wante to print a string forwards and backwards, as shown below:



```
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>
D:\Java\114-f03\examples\chap1_warmup>java warmup.AReverseUpperCasePrinter "Joh
 F. Kennedy"
Upper Case Letters:
JFK
Upper Case Letters in Reverse:
KFJ
D:\Java\114-f03\examples\chap1_warmup>_
```

Figure 3 Forward and Reverse Printing

We can scan the string twice, once forwards and once backwards, to implement this user interface, but that is inefficient. So what we will do instead is store the upper case characters we scan into an array, and then simply print the array backwards, as shown below:

```
package warmup;
public class AReverseUpperCasePrinter {
            static final int MAX_CHARS = 5;
            static char[] upperCaseLetters = new char[MAX_CHARS];
            static  int numberOfUpperCaseLetters = 0;
            public static void main(String[] args){
                        if (args.length != 1) {
                                    System.out.println("Illegal number of arguments:" + args.length + ". Terminating
program.");
                                    System.exit(-1);
                        }
                        int index = 0;
                        System.out.println("Upper Case Letters:");
                        while (index < args[0].length()) {
```

```
                if (Character.isUpperCase(args[0].charAt(index))) {
                    System.out.print(args[0].charAt(index));
                    storeChar(args[0].charAt(index));
                }
                index++;
            }
            System.out.println();
            printReverse();
    }
    public static void storeChar(char c) {
            if (numberOfUpperCaseLetters == MAX_CHARS) {
                System.out.println("Too many upper case letters. Terminating program. ");
                System.exit(-1);
            }
            upperCaseLetters[numberOfUpperCaseLetters] = c;
            numberOfUpperCaseLetters++;
    }
    public static void printReverse() {
            System.out.println("Upper Case Letters in Reverse:");
            for (int index =numberOfUpperCaseLetters - 1; index >= 0; index--) {
                System.out.print(upperCaseLetters[index]);
            }
    }

}
```
Figure 4 Code for Reverse and Forward Printing

In the interest of modularity, separate methods are defined for storing a character in the array and for printing them in reverse.

Comparison of two solutions

As we can see by comparing Figures 1 and 2, the second user-interface is an extension of the first one. Yet the programs implementing the two user-interfaces do not share any code. A symptom of this problem is that the main class does all of the computation. Ideally, a main class should be "skinny", simply instantiating classes and invoking methods in the instantiated classses. The real work should be carried out in the instantiated classes - that is what makes the code object-oriented and allows for code sharing

Let us look at some concrete consequences of violating this principle. Compare the two loops in the classes:

```
    //loop in AnUpperCasePrinter
    while (index < args[0].length()) {
            if (Chracater.isUpperCase(args[0].charAt(index)))
                System.out.print(args[0].charAt(index));
                index++;
            }
    }


    //loop in AReverseUpperCasePrinter
    while (index < args[0].length()) {
            if (isUpperCase(args[0].charAt(index))) {
                    System.out.print(args[0].charAt(index));
                    storeChar(args[0].charAt(index));
            }
            index++;
    }
```

The only difference in the two is that the latter does an extra step in the loop body of storing a character in the array. Yet we do not share the code.

If we followed our principle of creating an extra class for the scanning part, as shown in Figure 5, then we should be able to share code?

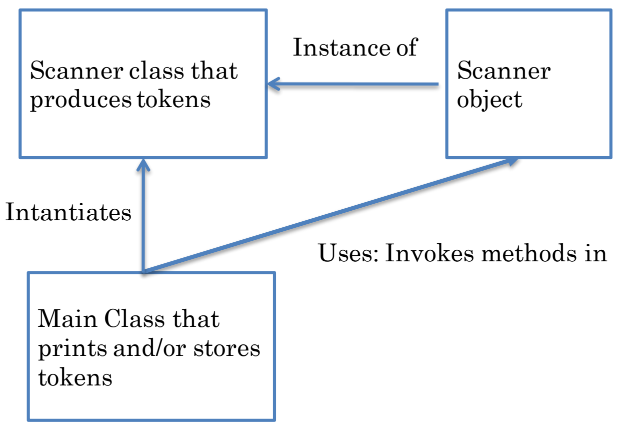

Scanner class that produces tokens
Instance of
Scanner object
Intantiates
Uses: Invokes methods in
Main Class that prints and/or stores tokens

Figure 5 Creating a separate scanner class

Index-based Scanner Interface

It is not easy to figure out the nature of the interface of such a class. One approach is to create one that exports an array of scanned tokens:

```
    public interface IndexBasedScanner {
            public String[] getTokenArray () ;
            public int getNumberOfTokens ();
    }
```

Here is an outline of the implementation of such an interface:

```
public class AnIndexBasedScanner implements IndexBasedScanner {
        static final int MAX_CHARS = 5;
        static char[] upperCaseLetters = new char[MAX_CHARS];
        static  int numberOfUpperCaseLetters = 0;
        public AnIndexBasedScanner(String theScannedString) {
                scanAndStore(theScannerString);
        }
        void scanAndStore (String theScannedString) {
                 // store all scanned tokens in upperCaseLetters
                …
        …
        public String[] getTokenArray () {
                return upperCaseLetters;
        }
        public int getNumberOfTokens {
                return numberOfUpperCaseLetters;
        }
}
```

The idea here is to extract and store all tokens when the scanner is instantiated. A user of the class simply accesses the stored values, which are exposed by two properties: *NumberOfTokens*, which stores the number of scanned tokens, and TokenArray, which stores the tokens in its first *NumberOfTokens* slots. The getter methods for these are trivial, simply returning values of associated instance variables. The only non-trivial method is *scan().* However, it is a straightforward variation of the code we saw above, scanning all tokens in a single loop and storing each of them in the array.

In comparison to the implementations we see below, this implementation is the simplest one. However, it has the serious disadvantage that it scans and stores all of the tokens, even if the user of the object needs only the first few tokens, and does no need these tokens to be stored.

To help us determine an interface, let us look at a real world problem involving radio scanning and at Java input libraries that do scanning.

Radio Scanning

Figure 6 shows how two parties can be involved in a scanning problem – one doing the scanning and the other using the results of the scanning.

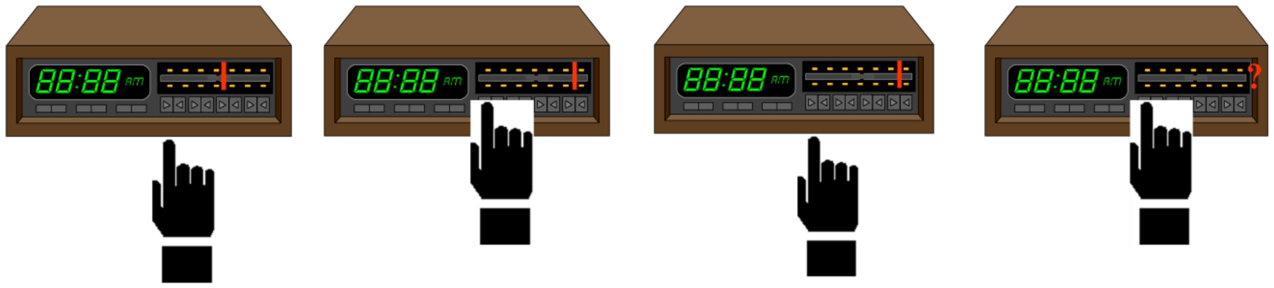


Figure 6 Analagy with Radio Scanning

Here the two parties are the human and a radio providing an automatic-scan operation. Each time the human invokes this operation, the radio automatically tunes to the next station. As the figure shows, we have to worry about invoking the operation when we reach the station broadcasting at the highest frequency. Modern radios simply restart the scan, tuning next to the station broadcasting at the lowest frequency. If such wrap-around was not provided, we would have to tell the user that no next station exists.

Consider the automatic-scan operation with a manual-scan operation, in which the human would have to manually tune the radio to the next frequency on which a station broadcasts. Here the human does all the work, much as, in the two scanning solutions above, the main class performs all of the scanning tasks.

Java Input using BufferedReader

Java input libraries shows how such division of labor in the real world can be translated into code. The following program illustrates the use of some of these libraries:

```
BufferedReader dataIn = new BufferedReader (new InputStreamReader(System.in));
int product = 1;
while (true) {
        int num = Integer.parseInt (dataIn.readLine());
        if (num < 0) break;
        product = product*num;
}
System.out.println (product);
```

This code is fairly straightforward. It processes input lines until a negative number is entered, converting each line to the corresponding number, and printing out the product of these numbers.

The code illustrates several aspects of input and object usage in general. The program fragment

```
dataIn.readLine()
```

executes the operation readLine() on the object stored in the variable dataIn. This operation returns the input string entered by the user on the next line. A user does not need to put quotes around the input string. This operation expects only strings, and considers the beginning and end of the input line as the string delimiters. Each time the method is called, it collects the next line from the user.

Let us now consider the more complicated program fragment:

```
new BufferedReader ( new InputStreamReader (System.in))
```

It creates a new instance of the predefined class `BufferedReader`, which is provided by Java for reading input. Java also provides the object, `System.in`, for reading input, which is counterpart of the object `System.out` provided for printing output. However, this object treats the console input as a sequence of bytes; and not a sequence of lines. The BufferReader instance converts the byte sequence created by System.in to a line sequence. It does so with the help of a newly created instance of InputStreamReader, which converts the byte stream to a character stream.

Thus, in the example above, we have three different objects working together to do string input. They are connected to each other as follows. The predefined object System.in is passed as a parameter to the instantiatiation of InputStreamReader in the code fragment:

```
new InputStreamReader (System.in))
```

The new instance of InputStreamReader in turn passed as a parameter to the instantiation of BufferedReader:

```
new BufferedReader ( new InputStreamReader (System.in)))
```

This instance can now read input lines from the console. Using three different objects to do input is consistent with the object-oriented philosophy of defining different objects for different aspects of a task and composing them to do the whole task. Here byte input, conversion of the input to a character stream, and conversion of the character stream to a line stream are done by three different classes chained together. This kind of object chaining happens in real life also – before ordering a house construction, you might order light fixtures, which you pass as a parameter to the house order.

Java Input as Scanning

The above program helps us understand how an independent scanner can be constructed for our two scanning problems involving printing of upper case letters. The Java libraries essentially convert an input byte stream into a line stream, as shown in Figure 6.

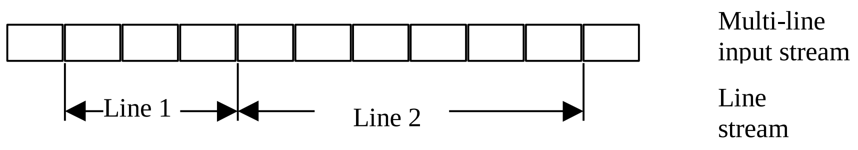


Figure 7 Scanning the Stream of Input Characters for Lines

Suppose this scanning task had been done in a while loop like the one above. In this case, the libraries would provide us with a loop in which tokens would be generated. Our task would be add processing code into this loop:

```
initialize
```

```
while there is more input
   set next line;
   application-specific code to process next line
   move next line markers
```

This means that all of our classes that process lines would have this program fragment. Moreover, as the library is updated, we would have to change all classes that have this code!

This is the reason why scanning is done as part of independent classes. In the code below:

```
DataInputStream dataIn = new BufferedReader (new InputStreamReader(System.in));
```

Two separarate scanners are used here, an instance of InputStreamReader and an instance of BufferedReader, which conver the byte stream representing user input to a character and line stream respectively. Each of these objects takes as a parameter to its constructor an object that produces the stream to be scanned. In case of the InputStreamReader this stream is the byte stream entered by the user, and in case of BufferedReader, this stream is the character stream produced by the InputStreamReader.

Now conside the BufferedReader object. This scanner produces elements of the token stream on demand, returning the next element each time the instance method `readLine()` is invoked on it. The first invocation returns the first token in the stream, and each subsequent invocation returns the token following the last one it returned.

Thus we see here how the concept of a scanner allows us to separate the parts of the program that scans for tokens and the part that processes them. The latter part simply instantiates the scanner, passing it the input stream in the constructor, and calls a method to repeatedly request successive elements of the output token stream.

If we could create a scanner for the input and token streams of our problem, then we would have a similar separation in our problem.

Iterator

Let us consider, first, the methods we should be able to invoke on the scanner. We need a method like `readLine()` that returns the next token. Our tokens are characters, not lines; thus the return type of this function is a `char` and not a `String`. We are not necessarily reading user input, simply asking for next element in the token stream; therefore, let us name it `nextElement` rather than `readElement`. The method declaration, then is:

```
public char next();
```

One of the problems with `dataIn` is that it does not provide a convenient way to determine if there in another line left in the token stream. It raises an `IOException` if we ask for another line and the user has terminated input. This is bad design, since exceptions should be raised on truly exceptional conditions, not on normal termination of input. To avoid using exceptions, our algorithm had to implement the notion of a sentinel value that terminated user input, a detail best left out of users of the scanner.

We will overcome this problem of `DataInputStream` by defining an explicit scanner operation that lets us know if there are any more elements left in the token stream:

```
        public boolean hasNext();
```

Thus, the scanner implements two operations: one that returns the next element in the token stream and another that indicates if the stream has more elements.

We can now define the interface of our scanner:

```
public interface CharIterator {
   public char next();
   public boolean hasNext();
}
```

Again, we are diverging from the design of the class `BufferedReader`. This class does not implement an interface, thereby not giving us the associated benefits such as ease of change.

The interface, `CharIterator,` is an example of an *iterator* (also called *enumeration*) interface. To better understand the nature of such an interface, here is another example of it, which produces string rather than char tokens

```
public interface StringIterator {
   public String next();
   public boolean hasNext();
}
```

Thus is the interface `BufferedReader` should have implemented.

An iterator interface defines an operation to return the next element in a stream and another to determine if there are more elements in the stream. Different Iterator interfaces differ in the type of the elements produced by them. We will use the same names for the two iterator operations in all iterator interfaces we create. Thus, the headers of these operations will be:

```
  public <Type> next();
  public boolean hasNext();
```

where <Type> is some Java type – primitive or object – defining elements of the token stream.

When we study Vectors, you will see another example of this interface, called `Iterator`, which is used extensively in Java.

Let us look more closely at `CharIterator`, the interface we must implement in this problem. It allows any character to be enumerated – not just upper case letters. The reason is that the type of the enumerated elements is `char`, which includes all characters – upper case letters, lowercase letters, numbers, space, and so on. It will be the responsibility of the implementation to restrict these characters to uppercase letters. Let us call the implementation, therefore, `AnUpperCaseIterator`. Like `BufferedReader`, its constructor takes the input stream as an argument, which in this example is a string Thus:

```
        new AnUpperCaseIterator (s)
```

creates a scanner that produces all the upper case letters in string `s`.

Using an Iterator

Before we define `AnUpperCaseIterator`, let us see how it (and the interface it implements) is used in the AnUpperCasePrinter the main class:

```
package main;
import enums.CharIterator;import enums.AnUpperCaseIterator;
public class UpperCasePrinter {
  public static void main (String args[]) {
    if (args.length != 1) {
        System.out.println("Illegal number of arguments:" + args.length + ". Terminating program.");
        System.exit(-1);
    }
    printUpperCase(args[0]);
  }

  public static void printUpperCase(String s) {
    System.out.println("Upper Case Letters:");
    printChars (new AnUpperCaseIterator(s));
  }
  public static void printChars (CharIterator charIterator) {
    while (charIterator.hasNext())
      System.out.print(charIterator.next());
  }
}
```

Figure 8 Iterator use

As before, we have been careful to separate the input and output code in the class. It is the output code that uses the Iterator. It creates a new instance of `AnUpperCaseIterator` for scanning the input string returned by `getInput()`. It then repeatedly invokes `next()` on the scanner to output the stream of upper case letters in the string. It stops when there are no more elements in the stream, that is, the method `hasNext()` returns false.

Similarly, the reverse upper case printer would use the scanner as follows:

```
while (charIterator.hasNext()) {
        char nextChar = charIterator.next());
        System.out.print(nextChar);
        storeChar(nextChar);
}
```

Since scanning is done in a separate class, both share this code.

We are now ready to develop the scanner implementation, `AnUpperCaseIterator`. The following sections discuss it in depth. However, to better appreciate the solution given here, it is best to first try to develop one on your own.

Two-Phase Storage-based Solution

It is possible to easily transform our existing scanner implementations to support the scanner interface. We can create a new scanner class whose constructor scans the characters in a while loop, and inserts each upper case character (token) in an array. Each call to next() simply accessed the appropriate element in the array. This solution is outlined below:

```
public class AnUpperCaseIterator implements CharIterator {
        static final int MAX_CHARS = 5;
        String[] upperCaseLetters;
        int numberOfUpperCaseLetters;
        ….
        public AnUpperCaseIterator(String theString) {{
                scanAndStore(theString);
        }
        void scanAndStore (String theString) {
                 // store all scanned tokens in array
                …
        }
        public  boolean  hasNext() {
                ….
        }
        public String next() {
                …
        }
}
```

This solution is much like the AnIndexBasedScanner we saw earlier. The main difference is that it implements an iterator interface rather than defining the TokenArray and NumberOfTokens properties. As a result, this implementation is more complex, as it must keep track of the next stored token that is to be returned by next() and if such a token exists. However, like AnIndexBasedScanner, it scans all tokens and stores them in an internal array. Thus, like it, it has the following disadvantages:

- *Less time efficient*: In this solution, all scanning occurs before first call to next(). As a result, the scanner object must find all tokens, even if the user of the object wishes to process only the first few tokens.
- Less space efficient: An array is needed to store all of the tokens, which can take a significant amount of space if the token stream is long.
- Less modular: All of the scanning is done in the single while loop executed by the constructor.

Iterator vs. Indexing

The discussion above allows us to compare iteration with array-like indexing. An iterator is an alternative to indexing for accessing elements of an ordered sequence of elements. It is somewhat more rigid than indexing in that the elements must be accessed sequentially from first to last. Indexing, on the other hand, allows *random access*, that is, allows us to access the elements in any order. Let us compare indexing and iteration when (some) elements are accessed in sequence.

The ability to do random accessing forces us to store all elements of the sequence in memory, which is both time and space efficient, as mentioned above. Moreover, for an application that wishes to access elements of the stream in sequence, without randomly accessing them, an iterator provides a simpler interface, as the application does not have to keep track of the index of the next element.

As we saw above, an iterator may, in fact, store all tokens in memory. However, unlike indexing, it does not require storage of all tokens. We see below a solution in which each call to next() scans, without storing, the next token.

Scanner Data Structures

The scanner's job is to scan each character of the string from left to right, looking for upper case letters. Let us first consider the variables or *data structures* it needs to do its job. Clearly, it needs the string to be scanned. It also needs a *marker* variable to tell it how much of the string it has scanned so far, that is, which characters it has already looked at.

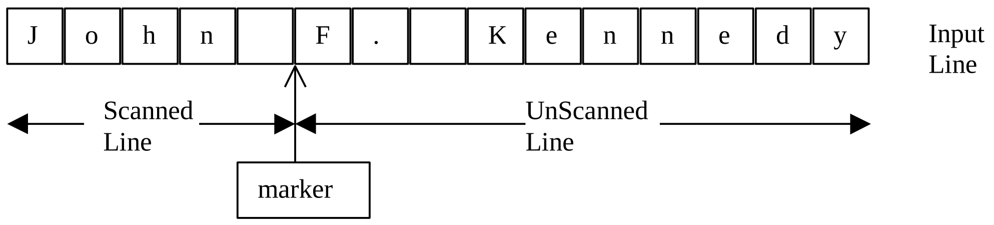


Figure 9 Scanner Marker

Every time it is asked for a new element in the token stream, the scanner advances the marker.

There is some flexibility in how much of the string is scanned when the scanner is asked for a new element. Two choices are:

- *At next element*: The string is scanned to the next element to be returned by the scanner. When the scanner is asked for the new element, it returns the token at the marked position and advances the marker to the next token. Thus, in the figure above, it would return the letter 'F' and advance the marker to 'K'.

- *At last element*: The string is scanned to the last element returned by the scanner. When the scanner is asked for the new element, it advances the marker to the next token and returns this token. Thus, in the example above, it advances the marker to the letter 'K' and returns 'K'.

These two approaches correspond to advancing a loop counter at the end or start of the loop body. In both approaches, when the marker is advanced, there may be no more elements left to return. In this case, we will let the marker go beyond the end of the input stream.

The first approach makes it easier to determine if a stream has more elements, since all we have to do is check if the marker is beyond the end of the input stream. The function, `hasNext()`, thus, can be a pure function without the side effect of changing the global marker variable. Therefore, we will use it in our solution, and call the marker variable `nextElementPos`, since it stores the position of the next element to be returned. If there is a next element, it stores the index of this element; otherwise it stores an index beyond the string.

Scanner Algorithms

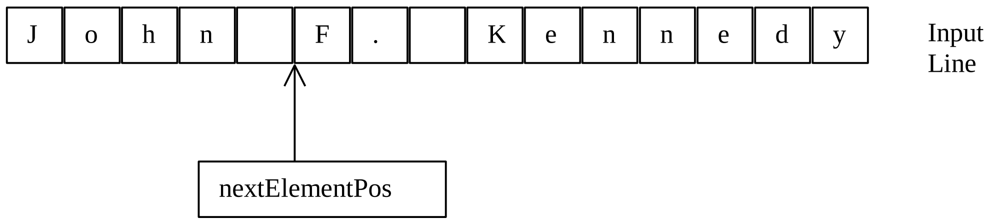
Now that we have identified the data structures of a class, let us next determine the algorithms used by the operations defined by the class. In general, when tacking a complex problem such as scanning a stream, it is best to first consider the data structures we need, then the algorithm, and finally the code.

The algorithm for `hasNext()` is straightforward. It must return true if nextElementPos is beyond end of string; false otherwise.

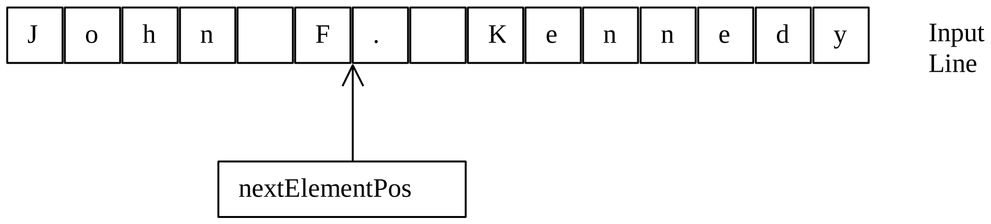
The algorithm for for `next()` is more complicated:

1. Return the element at `nextElementPos`.
2. Move `nextElementPos` beyond the element returned.
3. Skip to the next upper-case letter or end of string, whichever comes first.

To illustrate these steps, assume that `nexElementPos` is at 'F'.

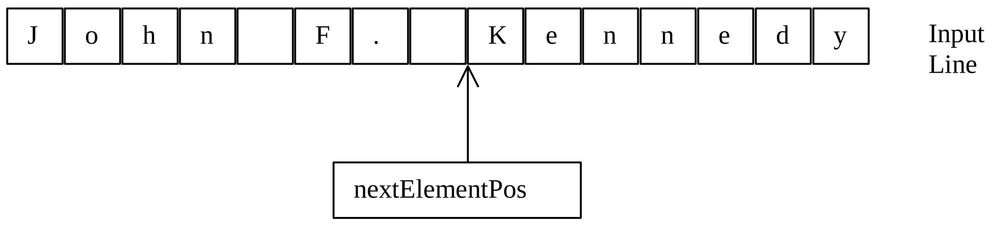


Consider what happens when the scanner is asked for the next element. It first returns 'F'. Next it advances the marker beyond the returned element:



All it did was increment `nextElementPos`, since the length of the returned token is 1 character. In general, it would need to determine the length of the token and add it to the marker.

If the next character were an uppercase letter, it would stop here. However, as we see here, an upper case letter may not be followed immediately by another upper case letter. Therefore, we must perform the third step of skipping non-uppercase letters.



The next time the scanner is asked for an element, it will return the upper case character at the new value of `nextElementPos`.

In this example, there was a remaining (unscanned) upper case letter in the string. If this was not the case, `next()` should skip beyond the end of the string, so that `hasNext()` can return false.

Scanner Code

We are finally ready to code the scanner class:

```
public class AnUpperCaseIterator implements CharIterator {
  String string;
  int nextElementPos = 0;
  public AnUpperCaseIterator(String theString) {
    string = theString;
    skipNonUpperCaseChars();
  }
  public boolean hasNext() {
    return nextElementPos < string.length();
  }
  public char next() {
    char retVal = string.charAt(nextElementPos);
    movePastCurrentElement();
    skipNonUpperCaseChars();
    return retVal;
  }
  void movePastCurrentElement() {
    nextElementPos++;
  }
 // keep advancing nextElementPos until we hit the next upper case or go
 // beyond the end of the string.
  void skipNonUpperCaseChars() {
    while (nextElementPos < string.length() && !Character.isUpperCase(string.charAt(nextElementPos)))
      nextElementPos++;
  }
}
```

Figure 10 Scanner Code

Each instance of this class enumerates its own stream of uppercase letters. The instance variable, `string`, stores the string being scanned, and the instance variable, `nextElementPos` is the marker for this string. Both variables are initialized by the constructor, which takes as an argument the string to be scanned. The constructor stores its argument in `string`, and advances `nextElementPos` to the first uppercase character. The other methods follow from the algorithm described earlier.

The function `next()` is impure – it returns the character at `nextElementPos` and also changes this variable. Thus, it performs the kind of side effect that is usually banned in functions – changing a global variable. This is an example of the kind of side effect `readLine()` also performs – returning a token from the stream and changing the token marker. Since, as a Java programmer, we are used to such a side effect when processing a stream, we make an exception for it – allowing it to change a global variable. As we know from using `readLine()`, it is convenient to call one method to perform both tasks, since, we rarely perform one without the other.

However, we must be careful to perform them in the correct order. When `next()` is called, `nextElementPos` is at the character the function must return. Therefore, we use the marker to retrieve the character before we calculate its new value. The retrieved character is stored in the variable `retVal`, which is then returned at the end of the function.

`next()` expects to find another element in the token stream. It is the job of the user of the Iterator to check that it `hasNext()` before calling `next()`.

To detect upper case letters, the class uses the method, `isUpperCase`, of the predefined Java class, `Character`, which returns true if its argument is an upper case letter.

Generalized Scanners**

The scanner class above is relatively simple because the token length is fixed. The following code outlines the kind of changes we would have to make to understand variable length tokens.

```
package <P>;
public class <Scanner Name> implements <T>Iterator {
   String string;
   int nextElementStart = 0;
   int nextElementEnd; // may not be global variable
   public <Scanner Name> (String theString) {
      string = theString;
      skipNonTokenCharacters();
   }
   public boolean hasNext() { return nextElementStart < string.length();}
   public <T>  next () {
      <T> retVal = extractToken();
      movePastCurrentToken(getLength(retVal));
      skipNonTokenCharacters();
      return retVal;
   }
   void movePastCurrentToken() {…};
   void skipNonTokenCharacters() {…};
   <T> extractToken() { …}
}
```

In these notes, identifiers within angle parameters such stand for application-specified abstract identifiers that take different concrete values in different applications. In the code above, <T> indicates an application specific type named T. In the upper case printer, it took the value char. Each instance of an abstract identifier takes or "unifies to" the same specific value. Thus, if we substitute one occurrence of <T> with char, we must do so for all occurrences.

As the code above shows, next() will, in general, return a token of some type <T>. As such a token can be of variable length, we will need to keep track of the end of the current token. Depending on the implementation, this value may be stored in local or global variable. After next() has found the current variable length token, it can pass the length of this token to movePastCurrentToken() so it can increment the marker.

What if the scanner produces multiple kinds of tokens such as words and numbers? In this case, our routines that implement different aspects of next() must be aware of the kind of the next token. For instance, if the next token is a number, then extractToken() must return a sequence of digits and if it is a word, then it must return a sequence of letters. To make code more modular and allow additional token kinds to be easily added, scanning of different kinds of tokens should be done in different methods. Which of these methods is called depends on the kind of the next token. It is usually possible to determine the kind of a token from its first character. For example, we know if the next token is a word or number based on whether the first character is a letter or digit.

The following code snippet illustrates how we can handle multiple token types:

```
 public <T>  next () {
    if isDigit(firstTokenChar)
          return nextNumber()
    else if isLetter(firstTokenChar)
         return nextWord()
 }
public <T>  nextNumber () {
    <T> retVal = extractNumber();
    movePastCurrentToken(getLength(retVal));
    skipNonTokenCharacters();
    return retVal;
 }
public <T>  nextWord() {
    <T> retVal = extractWord();
    movePastCurrentToken(getLength(retVal));
    skipNonTokenCharacters();
    return retVal;
 }
```

As the code shows, movePastCurrentToken() and skipNonTokenCharacters are token-independent.

Enumerating vs. Scanning

Scanning a stream is only one way to enumerate a sequence of values. The following, alternative, implementation of `CharIterator` illustrates that there are other ways:

```
public class AnotherUpperCaseIterator implements CharIterator {
   char nextLetter = 'A';
   public boolean hasNext() {
      return nextLetter <= 'Z';
   }
   public char next() {
      char retVal = nextLetter;
      nextLetter = (char) (nextLetter + 1);
      return retVal;
   }
}
```

Figure 11 Iterator Implementation without Scanning

An instance of this class enumerates all upper case letters. Instead of scanning characters in a string, it calculates them using their ordinal numbers. The first time `next()` is called, it returns ‘A’. Each subsequent time, it returns the character following the one it returned last time it was called. The variable `nextLetter` stores the letter to be returned by the next call to `next()`.

Once we have written this class, we can output all characters in the output stream by simply calling the `print` of the original slution, passing it an instance of this class:

```
print (new AnotherUppercaseIterator());
```

Recall that the print invokes the two scanner operations on its scanner argument to print all characters produced by it. We can reuse `print` because this method expects an instance of the type, `CharIterator`, which is implemented by `AnotherUppercaseIterator`.

More on Interface as Syntactic Specification

Unlike the case of other interfaces we have seen before such as BMISpreadsheet of Chapter 4, the two implementations of `CharIterator` have very different behaviors. In one case uppercase letters in a string are print, while in the other case all uppercase letters in the alphabet are print. One cannot substitute one implementation of an interface with the other and get the same result. All we are guaranteed is that the two implementations enumerate a sequence of characters. This is sufficient guarantee for the `print()` method, which does not really look at which characters are enumerated. It may not be for other methods. This further highlights the fact that an interface is just a syntactic specification of a type and not a semantic specification. To take the car analogy, two cars may have identical ways of pressing the horns, but emit different sounds when you press the horn. Some drivers may consider the horns equivalent because they are pressed in the same way, while others may not because of the differences in the sounds emitted. The important thing is that the same action is taken in both cars, which corresponds to writing polymorphic methods such as `print` that work with different implementations of an interface.

Iterator Design Pattern

We have seen several implementations of the concept of an iterator. All of them used the same naming conventions. Naturally, the concept is independent of the exact methods provides. For example, our iterator interface could have been defined as follows:

```
package enums;
public interface CharEnumeration{
   public char nextElement ();
   public boolean hasMoreElements());
}
```

Both naming conventions are, in fact, supported by Java libraries. This is why we have used the terms enumeration and iterator interchangeably. Even the routines provided by BufferedReader provide an iterator implementation. In general, we will say that an object is an iterator if it provides:

- A method that gives the next object is some object sequence.

- A method or exception that indicates there are no more objects in the sequence.

An iterator and iterator-user together define a reusable *design pattern* that can occur in several applications that need to access a sequence of objects.

Design Patterns

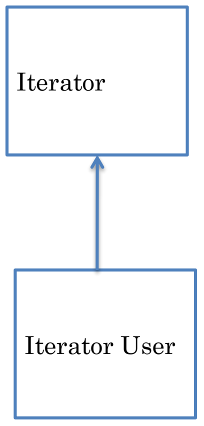


Figure 12 Iterator Design Pattern

Unlike a reusable interface or class, a design pattern is associated with multiple, in fact, infinite, classes and interfaces. It is independent of a particular programming language and is described informally, using diagrams, as we have done above. We have seen other language and class independent reusable patterns before, such as the notion of an enumerator, which can be mapped to multiple interfaces and classes, and a general way of creating enumerating scanners. Our previous patterns were mapped to single classes/interfaces while this design pattern involves two different classes. An Addison Wesley book, Elements of Reusable Object-Oriented Software, by Gamma, Helm, Johnson, and Vlissides, pioneered the notion of design patterns and is an encyclopaedia of important, multi class/interface design patterns, which includes the iterator design pattern. It is hard to read, and has been used mainly in graduate courses. In this course we will look at some other design patterns that our examples motivate in hopefully a more easy to digest manner.

The idea of a programming design pattern was inspired the notion of architectural patterns pioneered by Christopher. In his words:

- Each pattern is a three part rule, which expresses a relation between a certain context, a problem, and a solution.
- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of a solution to that problem, in such a way that you can use this solution a million times over

The same language can be used to describe a design pattern.

Scanner, Streams and Iterator

As we have seen in this chapter, scanners, streams and iterators are related to each other. As we have seen, a scanner can be an iterator, but an Iterator may not be a scanner. Based on the definition of an iterator above, a stream is a synonym for an iterator. Thus, InputStreamReader and BufferedReader are examples of iterators supporting program input. Unlike the objects we have defined in this section, these streams do not provide an explicit hasNext() operation. They also do not implement an interface, a problem with most Java classes early versions of Java (before Java 1.4). Java now defines an interface for user input, called java.io.Console. It also provides a special object, System.console that implements this interface by automatically composing System.in with BufferedReader.

Summary

- Indexing and enumerations are used to access instances of variable types. Iterators are easier to implement while indexing allows random access.

- A problem such as scanning that involves processing of a stream should have a separate object for enumerating the elements of the stream.

- When tacking a complex problem such as scanning a stream, it is best to first consider the data structures we need, then the algorithm, and finally the code.

- Loops that are both index-controlled and event-controlled must typically use a short-circuit Boolean operation in the loop condition to avoid subscript errors.

- If we do not declare a constructor in a class, Java automatically adds to the object code of the class an argument-less constructor with a null body.

- Different implementations of an interface can have very different external behaviors.

- A dynamic collection can be simulated by a named constant specifying the maximum size of the collection, a variable specifying the current size of the collection, an array for storing the elements of the collection.

- These three components of the collection should be encapsulated in a class and protected from direct external access by public methods.

Enumeration

1.