Lecture 17

Iterators

- Iterators and Iterables
- Chapter 8 with Annotations
- Lab 5 — Preview
  - PostScript Language
Iterators and Iterables
Three examples of iterating over a `Vector` of `Integers`. How do the last two examples work?

- A `Vector` is an *iterable* because it implements Java's `Iterator` interface, so it has an `iterator()` method, which returns an `Iterator` object that is used inside the `for` and `while` loops ...

  actually, the `AbstractIterator` interface from the `structure5` package extends the `Iterator` interface, and the `Vector`'s `iterator()` method returns a `VectorIterator` which implements `AbstractIterator`.

Let’s read the book!
Chapter 8 with Annotations
Chapter 8
Iterators

Concepts:
- Iterators
- The AbstractIterator class
- Vector iterators
- Numeric iteration

One potato, two potato, three potato, four, five potato, six potato, seven potato, more.
—A child’s iterator

Programs move from one state to another. As we have seen, this “state” is composed of the current value of user variables as well as some notion of “where” the computer is executing the program. This chapter discusses enumerations and iterators—objects that hide the complexities of maintaining the state of a traversal of a data structure.

Consider a program that prints each of the values in a list. It is important to maintain enough information to know exactly “where we are” at all times. This might correspond to a reference to the current value. In other structures it may be less clear how the state of a traversal is maintained. Iterators help us hide these complexities. The careful design of these control structures involves, as always, the development of a useful interface that avoids compromising the iterator’s implementation or harming the object it traverses.

- We have studied various container classes (e.g., arrays, lists, vectors).
- We often want to iterate over, or visit, every element in a container class once.
- Note that the text suggests that there will be an Iterator interface.
  - What would this interface contain?
8.1 Java’s Enumeration Interface

Java defines an interface called an Enumeration that provides the user indirect, iterative access to each of the elements of an associated data structure, exactly once. The Enumeration is returned as the result of calling the elements method of various container classes. Every Enumeration provides two methods:

```java
public interface java.utilEnumeration
{
    public abstract boolean hasMoreElements();
    // post: returns true iff enumeration has outstanding elements

    public abstract java.lang.Object nextElement();
    // pre: hasMoreElements
    // post: returns the next element to be visited in the traversal
}
```

- Enumeration is an interface that was in Version JDK1.0 of Java (early 1996) and it is very simple.
- It has mostly been superseded by the Iterator interface which was introduced in Version 1.2 (late 1998).
- The AbstractIterator interface in the structure package implements both.
The `hasMoreElements` method returns true if there are unvisited elements of the associated structure. When `hasMoreElements` returns false, the traversal is finished and the Enumeration expires. To access an element of the underlying structure, `nextElement` must be called. This method does two things: it returns a reference to the current element and then marks it visited. Typically `hasMoreElements` is the predicate of a while loop whose body processes a single element using `nextElement`. Clearly, `hasMoreElements` is an important method, as it provides a test to see if the precondition for the `nextElement` method is met.

The following code prints out a catchy phrase using a Vector enumeration:

```java
public static void main(String args[]) {
    // construct a vector containing two strings:
    Vector<String> v = new Vector<String>();
    v.add("Hello");
    v.add("world!");

    // construct an enumeration to view values of v
    Enumeration i = (Enumeration)v.elements();
    while (i.hasMoreElements())
        { // SILLY: v.add(1,"silly");
            System.out.println(i.nextElement()+" ");
        }
    System.out.println();
}
```

When run, the following immortal words are printed:

```
Hello world!
```
There are some important caveats that come with the use of Java's Enumeration construct. First, it is important to avoid modifying the associated structure while the Enumeration is active or live. Uncommenting the line marked SILLY causes the following infinite output to begin:

Hello silly silly silly silly silly silly

Inserting the string "silly" as the new second element of the Vector causes it to expand each iteration of the loop, making it difficult for the Enumeration to detect the end of the Vector.

**Principle 9** *Never modify a data structure while an associated Enumeration is live.*

Modifying the structure behind an Enumeration can lead to unpredictable results. Clearly, if the designer has done a good job, the implementations of both the Enumeration and its associated structure are hidden. Making assumptions about their interaction can be dangerous.

Another subtle aspect of Enumerations is that they do not guarantee a particular traversal order. All that is known is that each element will be visited exactly once before `hasMoreElements` becomes false. While we assume that our first example above will print out Hello world!, the opposite order may also be possible.

Presently, we develop the concept of an *iterator*.
### 8.2 The Iterator Interface

An Iterator is similar to an Enumerator except that the Iterator traverses an associated data structure in a predictable order. Since this is a behavior and not necessarily a characteristic of its interface, it cannot be controlled or verified by a Java compiler. Instead, we must assume that developers of Iterators will implement and document their structures in a manner consistent with the following interface:

```java
public interface java.util.Iterator
{
    public abstract boolean hasNext();
    // post: returns true if there is at least one more value to visit

    public abstract java.lang.Object next();
    // pre: hasNext()
    // post: returns the next value to be visited
}
```

While the Iterator is a feature built into the Java language, we will choose to implement our own AbstractIterator class.

```java
public abstract class AbstractIterator<E> implements Enumeration<E>, Iterator<E>, Iterable<E>
{
    public abstract void reset();
    // pre: iterator may be initialized or even mid-traversal
    // post: reset iterator to the beginning of the structure

    public abstract boolean hasNext();
    // post: true iff the iterator has more elements to visit

    public abstract E get();
    // pre: there are more elements to be considered; hasNext()
    // post: returns current value; ie. value next() will return

    public abstract E next();
    // pre: hasNext()
    // post: returns current value, and then increments iterator
```

- This courses uses data structure from standard Java’s libraries and the structure package.
- The structure package mirrors parts of Java’s libraries for educational and practical purposes.
- AbstractIterator is a nice extension of Java’s structures.
This abstract base class not only meets the `Iterator` interface, but also implements the `Enumeration` interface by recasting the `Enumeration` methods in terms of `Iterator` methods. We also provide some important methods that are not part of general `Iterators`: `reset` and `get`. The `reset` method reinitializes the `AbstractIterator` for another traversal. The ability to traverse a structure multiple times can be useful when an algorithm makes multiple passes through a structure to perform a single logical operation. The same functionality can be achieved by constructing a new `AbstractIterator` between passes. The `get` method of the `AbstractIterator` retrieves a reference to the `current element` of the traversal. The same reference will be returned by the call to `next`. Unlike `next`, however, `get` does not push the traversal forward. This is useful when the current value of an `AbstractIterator` is needed at a point logically distant from the call to `next`.

The use of an `AbstractIterator` leads to the following idiomatic loop for traversing a structure:

```java
public static void main(String args[]) {
    // construct a vector containing two strings:
    Vector<String> v = new Vector<String>();
    AbstractIterator<String> i;
    v.add("Hello");
    v.add("World!");

    // construct an iterator to view values of v
    for (i = (AbstractIterator<String>)v.iterator(); i.hasNext(); i.next()) {
        System.out.print(i.get()+" ");
    }
    System.out.println();
}
```

The result is the expected `Hello World!`
8.3 Example: Vector Iterators

For our first example, we design an Iterator to traverse a Vector called, not surprisingly, a VectorIterator. We do not expect the user to construct Vector-Iterators directly—instead the Vector hides the construction and returns the new structure as a generic Iterator, as was seen in the HelloWorld example. Here is the iterator method:

```java
public Iterator<E> iterator()
// post: returns an iterator allowing one to
// view elements of vector
{
    return new VectorIterator<E>(this);
}
```

When a Vector constructs an Iterator, it provides a reference to itself (this) as a parameter. This reference is used by the VectorIterator to recall which Vector it is traversing.

We now consider the interface for a VectorIterator:
The outside world will not have access to `VectorIterator`. It belongs to the structure package and is not public.

The outside world will treat every `VectorIterator` object as if it were simply an `Iterator` object.

As is usually the case, the nonconstructor methods of `VectorIterator` exactly match those required by the `Iterator` interface. Here is how the `VectorIterator` is constructed and initialized:

```java
protected Vector<E> theVector;
protected int current;

public VectorIterator(Vector<E> v)
// post: constructs an initialized iterator associated with v
{
    theVector = v;
    reset();
}

public void reset()
// post: the iterator is reset to the beginning of the traversal
{
    current = 0;
}
```
The constructor saves a reference to the associated Vector and calls reset. This logically attaches the Iterator to the Vector and makes the first element (if one exists) current. Calling the reset method allows us to place all the resetting code in one location.

To see if the traversal is finished, we invoke hasNext:

```java
public boolean hasNext()
// post: returns true if there is more structure to be traversed
{
    return current < theVector.size();
}
```

This routine simply checks to see if the current index is valid. If the index is less than the size of the Vector, then it can be used to retrieve a current element from the Vector. The two value-returning methods are get and next:

```java
public E get()
// pre: traversal has more elements
// post: returns the current value referenced by the iterator
{
    return theVector.get(current);
}

public E next()
// pre: traversal has more elements
// post: increments the iterated traversal
{
    return theVector.get(current++);
}
```

The get method simply returns the current element. It may be called arbitrarily many times without pushing the traversal along. The next method, on the other hand, returns the same reference, but only after having incremented current. The next value in the Vector (again, if there is one) becomes the current value.
Since all the Iterator methods have been implemented, Java will allow a VectorIterator to be used anywhere an Iterator is required. In particular, it can now be returned from the iterator method of the Vector class.

Observe that while the user cannot directly construct a VectorIterator (it is a nonpublic class), the Vector can construct one on the user's behalf. This allows measured control over the agents that access data within the Vector. Also, an Iterator is a Java interface. It is not possible to directly construct an Iterator. We can, however, construct any class that implements the Iterator interface and use that as we would any instance of an Iterator.

Since an AbstractIterator implements the Enumeration interface, we may use the value returned by Vector's iterator method as an Enumeration to access the data contained within the Vector. Of course, treating the VectorIterator as an Enumeration makes it difficult to call the AbstractIterator methods reset and get.
8.4 Example: Rethinking Generators

In Section 7.2 we discussed the construction of a class of objects that generated numeric values. These Generator objects are very similar to Abstract-Iterators—they have next, get, and reset methods. They lack, however, a hasNext method, mainly because of a lack of foresight, and because many sequences of integers are infinite—their hasNext would, essentially, always return true.

Generators are different from Iterators in another important way: Generators return the int type, while Iterators return Objects. Because of this, the Iterator interface is more general. Any Object, including Integer values, may be returned from an Iterator.

In this section we experiment with the construction of a numeric iterator—a Generator-like class that meets the Iterator interface. In particular, we are interested in constructing an Iterator that generates prime factors of a specific integer. The PFIterator accepts the integer to be factored as the sole parameter on the constructor:

```
import structure5.AbstractIterator;
public class PFGenerator extends AbstractIterator<Integer>
{
    // the original number to be factored
    protected int base;

    public PFGenerator(int value)
    // post: an iterator is constructed that factors numbers
    {
        base = value;
        reset();
    }
}
```
The process of determining the prime factor involves reducing the number by a factor. Initially, the factor $f$ starts at 2. It remains 2 as long as the reduced value is even. At that point, all the prime factors of 2 have been determined, and we next try 3. This process continues until the reduced value becomes 1.

Because we reduce the number at each step, we must keep a copy of the original value to support the reset method. When the iterator is reset, the original number is restored, and the current prime factor is set to 2.

```java
// base, reduced by the prime factors discovered
protected int n;
// the current prime factor
protected int f;

public void reset()
// post: the iterator is reset to factoring the original value
{
    n = base;
    // initial guess at prime factor
    f = 2;
}
```

If, at any point, the number $n$ has not been reduced to 1, prime factors remain undiscovered. When we need to find the current prime factor, we first check to see if $f$ divides $n$—if it does, then $f$ is a factor. If it does not, we simply increase $f$ until it divides $n$. The next method is responsible for reducing $n$ by a factor of $f$.
public boolean hasNext()
// post: returns true iff there are more prime factors to be considered
{
    return f <= n;  // there is a factor <= n
}

public Integer next()
// post: returns the current prime factor and "increments" the iterator
{
    Integer result = get();  // factor to return
    n /= f;  // reduce n by factor
    return result;
}

public Integer get()
// pre: hasNext()
// post: returns the current prime factor
{
    // make sure f is a factor of n
    while (f <= n && n%f != 0) f++;
    return f;
}

● Notes
We can now write a program that uses the iterator to print out the prime factors of the values presented on the command line of the Java program as it is run:

```java
public static void main(String[] args) {
    // for each of the command line arguments
    for (int i = 0; i < args.length; i++)
        {
            // determine the value
            int n = Integer.parseInt(args[i]);
            PFGenerator g = new PFGenerator(n);
            System.out.print(n+" : ");
            // and print the prime factors of n
            while (g.hasNext()) System.out.print(g.next()+" ");
            System.out.println();
        }
}
```

For those programmers that prefer to use the `hasMoreElements` and `next-Element` methods of the `Enumeration` interface, those methods are automatically provided by the `AbstractIterator` base class, which `PFGenerator` extends.
Exercise 8.1 The $3n + 1$ sequence is computed in the following manner. Given a seed $n$, the next element of the sequence is $3n + 1$ if $n$ is odd, or $n/2$ if $n$ is even. This sequence of values stops whenever a 1 is encountered; this happens for all seeds ever tested. Write an Iterator that, given a seed, generates the sequence of values that ends with 1.

- This is in reference to the Collatz Conjecture.
- Recommended: Try doing this exercise yourself if you haven’t already done so.
- We’ll work on it together next class.
Lab 5 — Preview
PostScript Language
PostScript

PostScript is a text file, a vector graphics file format, and also a programming language. It was invented by Adobe in the early 1980s.

The image (left) and program (right) are two different ways of interpreting the same .ps file. Image by Aaron Santiago while at Simon’s Rock.

In Lab 5, you’ll help write a partial non-graphical interpreter for PostScript. All of the needed information is in the lab handout. The following slides add bonus motivation and context.
Graphic (left) and the code (right) that is used to draw it. In reality, they are the same artifact (i.e., the same text file) with different interpretations.
Change the 4s to 5s and get a new image when you open it again.
Programming Tools

There are two primary tools for programming in PostScript:

- Ghostscript. This is a program for interpreting the PostScript language.
- Ghostview. This is a program for viewing PostScript files as images.

These tools were first released in 1988 (see Wikipedia).

These programs are available in our department’s Unix environment. In particular, you’ll use ghostscript via `gs -DNODISPLAY` in the lab.
Infix Notation

We typically write formulae using *infix* notation. This means that the binary operators are written between its two operands.

\[
1 + 2 = 3
\]

\[
1 + 2 \times 3 = 1 + (2 \times 3) = 1 + 6 = 7
\]

\[
(1 + 2) \times 3 = 3 \times 3 = 9
\]

Computing requires order of operations, and brackets manipulate this order.

Question: How would we write a ternary operator using infix notation.

One solution is to use two symbols for operators as in Java’s ternary operator.
Polish Notation

In Polish Notation the operation is written before its two operands.

\[
\begin{align*}
+ \ 1 \ 2 &= 3 \\
+ \ * \ 2 \ 3 \ 1 &= + \ 6 \ 1 = 7 \\
* \ + \ 1 \ 2 \ 3 &= * \ 3 \ 3 = 9 \\
\end{align*}
\]

Notice that brackets are no longer necessary.

Also, ternary operations can be handled in the same way.

\[
\text{operation} \ \text{operand}_1 \ \text{operand}_2 \ \text{operand}_3
\]
Reverse Polish Notation (RPN)
In Reverse Polish Notation the operation is written after its two operands.

\[ 1 \ 2 \ + \ = \ 3 \]
\[ 2 \ 3 \ * \ 1 \ + \ = \ 6 \ 1 \ + \ = \ 7 \]
\[ 1 \ 2 \ + \ 3 \ * \ = \ 3 \ 3 \ * \ = \ 9 \]

Most people find this notation to be more natural than Polish Notation. It can also be easier for parsing programs.

This is how PostScript and other stack-based languages (e.g. Forth) operate.
# Evaluating Arithmetic in RPN

Arithmetic expressions in RPN can be evaluated using a stack. The stack is initially empty and the expression is read from left-to-right.

- Values are pushed onto the stack.
- Binary operators are evaluated by popping the top two values off of the stack.

**Example:** Evaluate \(1 \ 2 \ 3 \ + \ 4 \ + \ *.\)

<table>
<thead>
<tr>
<th>Stack</th>
<th>Expression</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The answer is the only thing on the stack after the expression is computed.

- Popping an empty stack occurs if the expression is not well-formed.
- Terminating with multiple values on the stack occurs if the expression is not well-formed.
Variables

Variables are created and referenced and redefined as follows.

- There are no keywords in PostScript so be careful!

```
/r 0.5 def
/g 0.5 def
/b 0.5 def
r g b
setrgbcolor

/x 1 def  % x is now 1
/x 2 def  % x is now 2
/x x 1 add def  % x is now 3

/def 5 def  % don't do this!
/def 5 def  % aaahhhhh!
```

Variables are actually entries in a dictionary.

- A name is created by / followed by characters do not comprise a number.
- The value of /name is accessed using name (without the slash).
- Name values are stored in dictionaries, which will be discussed later.
Evaluating Functions in RPN

The same principle can be applied to evaluating programs written in RPN. The stack is initially empty and the program is read from left-to-right.

- Parameters are pushed onto the stack.
- k-ary functions are evaluated by popping the top k values off of the stack.

**Example**: Suppose \( \text{max} \) returns the maximum of 3 arguments and \( \text{neg} \) negates one argument.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 neg 3 max</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1 1 1 3</td>
</tr>
</tbody>
</table>

Again the stack terminates with the answer iff the program is well-formed. This method of computing is extremely efficient.
PostScript Printers

You may have noticed that some printers identify as *PostScript printers*. These printers run PostScript programs when they print your documents.

What is going on in there?

What are the pros and cons of running programs for print jobs?

- File size. Programs are often smaller.
- Scalability. Vector graphics can be scaled without loss of quality.
- Security. Malicious code can be embedded into PostScript files or printers.
- Reliability. What if there is an infinite loop in a file being printed?
PostScript Fonts

PostScript fonts store the outline of each character in the PostScript language.

An individual letter in a PostScript font

The PostScript languages allows arbitrary linear transformations (rotation, scaling, translation) without any loss of quality. Hence, PostScript fonts can be perfectly rendered at any point size, orientation, etc.
Desktop Publishing

PostScript was the file format behind the early advances vector graphic editors and desktop publishing.

Encapsulated PostScript (.eps) is one of the most widely used formats by publishers. These files can include a bitmap preview of the image, which allows programs to show the image without running the included PostScript image.
Portable Document Format (PDF)

Adobe created the portable document format (pdf) based-off of the PostScript format. These files contain three parts:

- A subset of the PostScript language.
- A font embedding system which allows pdf files to contain fonts.
- A method for storing and compressing various elements into a single file.

Here are some YouTube links ([link1](#), [link2](#), [link3](#)) for videos on the history of PostScript / pdf.
Stack Operations

PostScript programs are easier and faster when they avoid creating many variables. Instead use the stack for saving values and storing intermediate calculations.

<table>
<thead>
<tr>
<th>% special value mark</th>
<th>20   % 20 is on top</th>
<th>% rotate the top n values % upward k positions n k roll</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>10     % 10 is on top</td>
<td>% equivalent to exch 2 1 roll</td>
</tr>
<tr>
<td>% count the values</td>
<td>exch    % 10 and 20 switch</td>
<td></td>
</tr>
<tr>
<td>% above the mark,</td>
<td>dup     % another copy of 20</td>
<td></td>
</tr>
<tr>
<td>% then clear them</td>
<td>pop      % one copy removed</td>
<td></td>
</tr>
<tr>
<td>counttomark</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cleartomark</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

It is helpful to practice these commands with the interactive \texttt{gs} shell.

- The \texttt{pstack} command properly prints out the \texttt{mark} values.
Functions

Functions can be defined and called as follows.

```
/setDataGrey
{
  /r 0.5 def
  /g 0.5 def
  /b 0.5 def
  r g b setrgbcolor
}
bind def
setDataGrey
```

```
/setMediumGrey
{
  0.5 0.5 0.5 setrgbcolor
}
bind def
setMediumGrey
```

Functions are defined and called using names whose values are code blocks.

- Many features of other languages (i.e. automatic local variables) are not present in PostScript functions.
- A code block `{ ... }` is actually an executable array, as discussed later.

The `bind` keyword replaces the value of each name with its current value.
Boolean Values and Relational Operators

Boolean values and relational values are illustrated below.

```
/b true def

1 1 eq   % results in true being pushed onto the stack
1 2 lt   % results in true being pushed onto the stack
1 1 lt   % results in false being pushed onto the stack
1 1 le   % results in true being pushed onto the stack

b false and % results in false being pushed onto the stack
b false or  % results in true being pushed onto the stack
```

Strings can also be compared using these relational operators.
Text Literals

String literals are created as follows.

(string)
(string with (parentheses\) inside)
(string with \b backslash inside)
(string with \n new line)
(string with character code 100 in octal \100)

/s 10 string % this creates a blank string with 10 characters
If Statements

If statements and if/else statements are created by the following code.

```
x 1 eq 
  { 
    % code block 
  } 
if

x 1 eq 
  { 
    % true block 
  } 
  { 
    % false block 
  } 
ifelse
```
Loops
Loops can be created in three ways.
However, they won’t be discussed in the lab.

Note: The for loop is the only one of the loops that modifies the stack directly. Each successive value of the loop counter goes on the stack before running the code block. Thus, \texttt{1 2 10 \{ \}} \texttt{for} completes with \texttt{1 3 5 7 9} on the stack.