Lecture 17

Iterators

- Iterators and Iterables
- Chapter 8 with Annotations
- Lab 5 – Preview
  - PostScript Language

Iterators and Iterables

```
  GNU nano 4.8                     SimpleIterator.java
import structure5.*;

public class SimpleIterator {

    public static void main(String[] args) {

        Vector<Integer> v = new Vector<>();
        v.add(10);
        v.add(20);
        v.add(30);

        example0(v);
        example1(v);
        example2(v);
    }
```

```
    public static void example0(Vector<Integer> v) {

        for (int index = 0; index < v.size(); index++) {
            System.out.println( v.get(index) );
        }
    }
```

```
    public static void example1(Vector<Integer> v) {

        for (Integer temp : v) {
            System.out.println(temp);
        }
    }
```

```
    public static void example2(Vector<Integer> v) {

        Iterator<Integer> i = v.iterator();
        Integer temp;

        while (i.hasNext()) {
            temp = i.next();
            System.out.println(temp);
        }
    }

}
```

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:

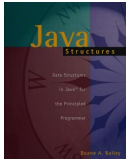
- ▷ Iterators
- ▷ The `AbstractIterator` class
- ▷ `Vector` iterators
- ▷ Numeric iteration

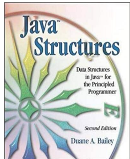
One potato, two potato, three potato, four,
five potato, six potato, seven potato, more.
—A child's iterator

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?
-



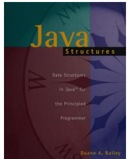


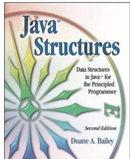
8.1 Java's Enumeration Interface

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:

```
public interface java.util.Enumeration
{
    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.

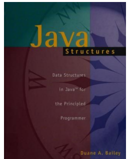
The following code prints out a catchy phrase using a `Vector` enumeration:

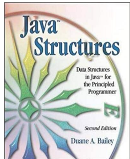
```
public static void main(String args[])
{
    // construct a vector containing two strings:
    Vector<String> v = new Vector<String>();
    v.add("Hello");
    v.add("world!");

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

When run, the following immortal words are printed:

```
Hello world!
```

- The comment "marks it visited" should not be taken completely literally.
- It looks like the `Vector` class must have an `elements()` method.
  - Look in `Vector.java` ...
  - Look in `AbstractList.java` ...
  - Look in `List.java` ...
  - Look in `Structure.java` ...
- The `iterator()` method has similar ancestry.





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.*

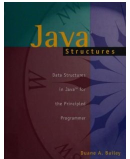
Modifying the structure behind an `Enumeration` can lead to unpredictable results. Clearly, if the designer has done a good job, the implementations of both

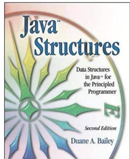
the `Enumeration` and its associated structure are hidden. Making assumptions about their interaction can be dangerous.

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*.

- Notes





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:

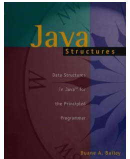
```
public interface java.util.Iterator
{
    public abstract boolean hasNext();
    // post: returns true if there is at least one more value to visit

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

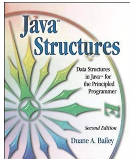
While the `Iterator` is a feature built into the Java language, we will choose to implement our own `AbstractIterator` class.

```
public abstract class AbstractIterator<E>
    implements Enumeration<E>, Iterator<E>, Iterable<E>
{
    public abstract void reset();
    // pre: iterator may be initialized or even amid-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.

```
    public void remove()
    // pre: hasNext() is true and get() has not been called
    // post: the value has been removed from the structure
    {
        Assert.fail("Remove not implemented.");
    }

    final public boolean hasMoreElements()
    // post: returns true iff there are more elements
    {
        return hasNext();
    }

    final public E nextElement()
    // pre: hasNext()
    // post: returns the current value and "increments" the iterator
    {
        return next();
    }

    final public Iterator<E> iterator()
    // post: returns this iterator as a subject for a for-loop
    {
        return this;
    }
```





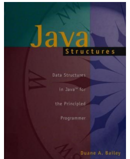
This abstract base class not only meets the `Iterator` interface, but also implements the `Enumeration` interface by recasting the `Enumeration` methods in terms of `Iterator` methods. We also provide some important methods that are not part of general `Iterators`: `reset` and `get`. The `reset` method 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`.

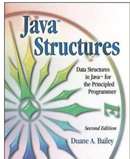
The use of an `AbstractIterator` leads to the following idiomatic loop for traversing a structure:

```
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!`

- Notes





8.3 Example: Vector Iterators

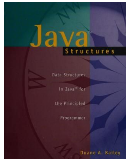
For our first example, we design an `Iterator` to traverse a `Vector` called, not surprisingly, a `VectorIterator`. We do not expect the user to construct `VectorIterators` directly—instead the `Vector` hides the construction and returns the new structure as a generic `Iterator`, as was seen in the `HelloWorld`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

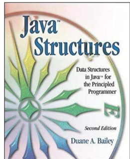
```
public Iterator<E> iterator()
// post: returns an iterator allowing one to
//       view elements of vector
{
    return new VectorIterator<E>(this);
}
```

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`:

- Notes



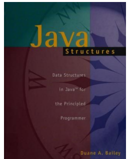


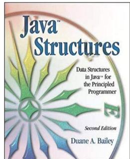
```
class VectorIterator<E> extends AbstractIterator<E>
{
    public VectorIterator(Vector<E> v)
    // post: constructs an initialized iterator associated with v

    public void reset()
    // post: the iterator is reset to the beginning of the traversal

    public boolean hasNext()
    // post: returns true if there is more structure to be traversed

    public E get()
    // pre: traversal has more elements
    // post: returns the current value referenced by the iterator

    public E next()
    // pre: traversal has more elements
    // post: increments the iterated traversal
}
```

- 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:

```
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`:

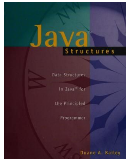
```
public boolean hasNext()
// post: returns true if there is more structure to be traversed
{
    return current < theVector.size();
}
```

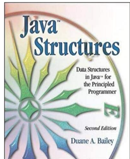
This routine simply checks to see if the current index is valid. If the index is less than the size of the `Vector`, then it can be used to retrieve a current element from the `Vector`. The two value-returning methods are `get` and `next`:

```
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.

- Notes



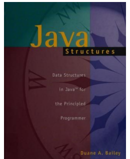


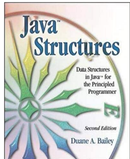
Since all the `Iterator` methods have been implemented, Java will allow a `VectorIterator` to be used anywhere an `Iterator` is required. In particular, it can now be returned from the `iterator` method of the `Vector` class.

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 `Vector-Iterator` as an `Enumeration` makes it difficult to call the `AbstractIterator` methods `reset` and `get`.

- Notes





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 `AbstractIterators`—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`.

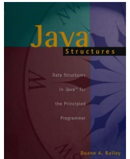
`Generators` are different from `Iterators` in another important way: `Generators` return the `int` type, while `Iterators` return `Objects`. Because of this,

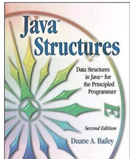
the `Iterator` interface is more general. Any `Object`, including `Integer` values, may be returned from an `Iterator`.

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 usage of `Generator` is somewhat unfortunate in the textbook.
- Java added a concept called a *generator* in Java 8 which is used by `Stream` objects and the `generate` method.





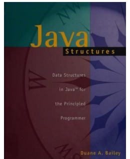
The process of determining the prime factor involves reducing the number by a factor. Initially, the factor `f` starts at 2. It remains 2 as long as the reduced value is even. At that point, all the prime factors of 2 have been determined, and we next try 3. This process continues until the reduced value becomes 1.

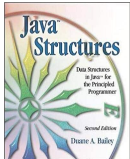
Because we reduce the number at each step, we must keep a copy of the original value to support the `reset` method. When the iterator is reset, the original number is restored, and the current prime factor is set to 2.

```
// 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`.

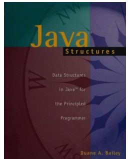
- This slide and the next two slides provide a really nice example.

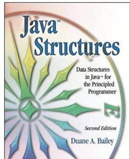




```
public boolean hasNext()
// post: returns true iff there are more prime factors to be considered
{
    return f <= n;          // there is a factor <= n
}

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



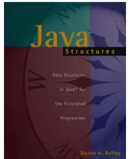


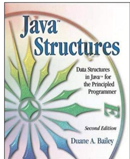
We can now write a program that uses the iterator to print out the prime factors of the values presented on the command line of the Java program as it is run:

```
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 `nextElement` methods of the `Enumeration` interface, those methods are automatically provided by the `AbstractIterator` base class, which `PFGenerator` extends.

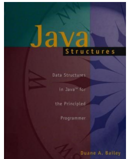
- Notes

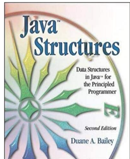




Exercise 8.1 *The* $3n + 1$ *sequence is computed in the following manner. Given a seed* n*, the next element of the sequence is* $3n + 1$ *if* n *is odd, or* $n/2$ *if* n *is even. This sequence of values stops whenever a* 1 *is encountered; this happens for all seeds ever tested. Write an* `Iterator` *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.





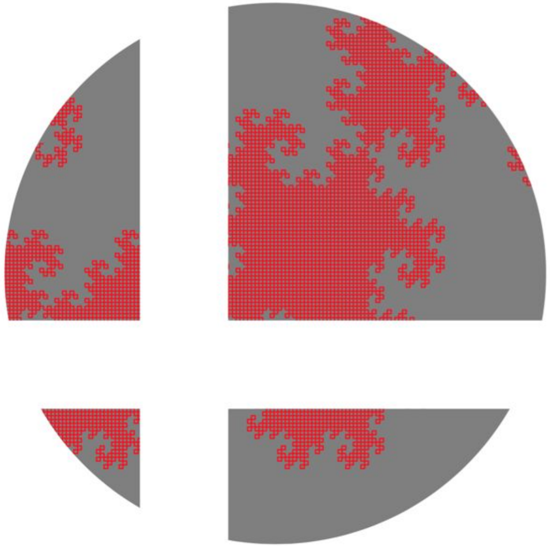
Lab 5 – Preview

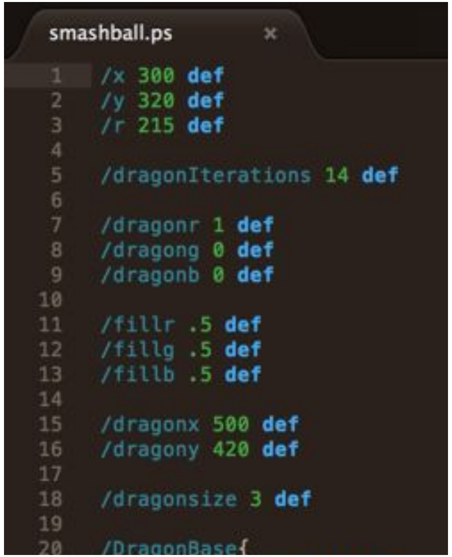
PostScript Language

PostScript

PostScript is a text file, a vector graphics file format, and also a programming language.

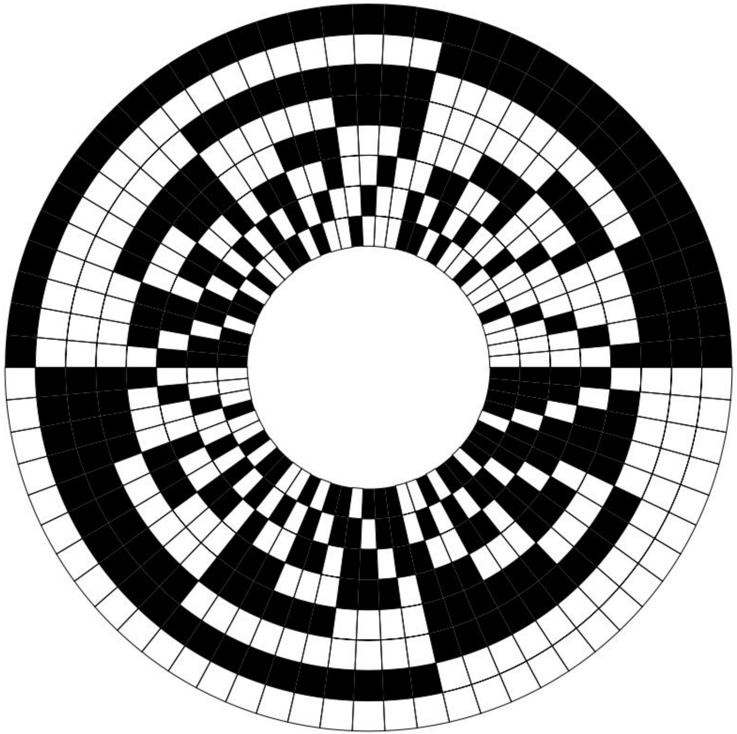
It was invented by Adobe in the early 1980s.





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.

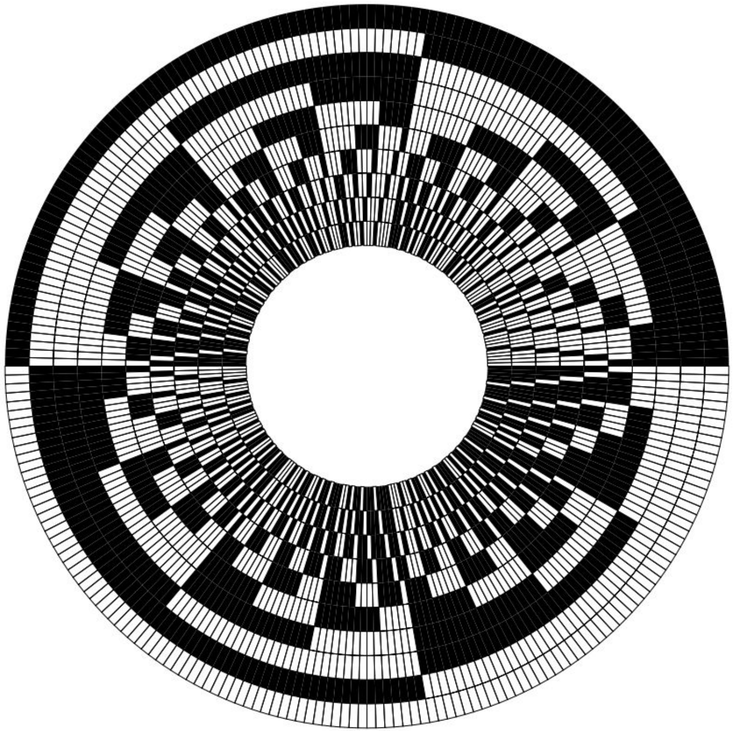
In Lab 5, you'll help write a partial non-graphical interpreter for PostScript. All of the needed information is in the lab handout. The following slides add bonus motivation and context.



```
% Generates the (S,T)-combinations in cool-lex

/S 4 def
/T 4 def

S T colex


% Draws a bitwheel containing M bitstrings of N
% The wheel has visible radius is V, and total
% The lines are drawn with width W points.
% Bits of value one are drawn by the ONE functi
% bits of value zero are drawn by the ZERO func
% (The recommended functions are {fill}, {strok
% Use {} and then N R V drawrings if you just w
% Use bitwheel (regular) or bitwheelinout (insi

/N S T add def
/M N T choose def
/R 1.5 inch def
/V 1 inch def
/W 0 def
/ONE {fill} def
/ZERO {stroke} def

% N R V W drawrings
N M R V W {ONE} {ZERO} bitwheelinout
```

```
% run the algorithm
1 1 m 1 sub {
  pop

  % col begins 0^x1^y0
  % count leading 0s
  /x 0 def
  {
    col x get 0 eq
    { /x x 1 add def }
    { exit }
    ifelse
  } loop

  % then count 1s
  /y 0 def
  {
    col x y add get 1 eq
    { /y y 1 add def }
    { exit }
    ifelse
  } loop

  % write 1^{y-1}0^{x+1}1 to the beginning of col
  0 1 y 2 sub {
    col swap2 1 put
  } for
  y 1 sub 1 x y add 1 sub {
    col swap2 0 put
  } for
  col x y add 1 put
```

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.



```
% Generates the (S,T)-combinations in cool-lex order.

/S 4 def
/T 4 def

S T colex


% Draws a bitwheel containing M bitstrings of N bits.
% The wheel has visible radius is V, and total radius R.
% The lines are drawn with width W points.
% Bits of value one are drawn by the ONE function, and
% bits of value zero are drawn by the ZERO function.
% (The recommended functions are {fill}, {stroke}, or {}).
% Use {} and then N R V drawrings if you just want concentric rings
% Use bitwheel (regular) or bitwheelinout (inside-out drawing)

/N S T add def
/M N T choose def
/R 1.5 inch def
/V 1 inch def
/W 0 def
/ONE {fill} def
/ZERO {stroke} def

% N R V W drawrings
N M R V W {ONE} {ZERO} bitwheelinout
```

Change to 5s

Try coollex

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.

$$1 + 2 = 3$$

$$1 + 2 * 3 = 1 + (2 * 3) = 1 + 6 = 7$$

$$(1 + 2) * 3 = 3 * 3 = 9$$

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.

$$+ \; 1 \; 2 = 3$$

$$+ \; * \; 2 \; 3 \; 1 = + \; 6 \; 1 = 7$$

$$* \; + \; 1 \; 2 \; 3 = * \; 3 \; 3 = 9$$

Notice that brackets are no longer necessary.

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

$$\textit{operation} \;\; \textit{operand}_1 \;\; \textit{operand}_2 \;\; \textit{operand}_3$$

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 + *.

				3		4		
			2	2	5	5	9	
Stack		1	1	1	1	1	1	9
Expression	1 2 3 + 4 + *	2 3 + 4 + *	3 + 4 + *	+ 4 + *	4 + *	+ *	*	

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`
		`/if 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 `max` returns the maximum of 3 arguments and `neg` negates one argument.

					3	
			2	-2	-2	
Stack		1	1	1	1	3
Expression	`1 2 neg 3 max`	`2 neg 3 max`	`neg 3 max`	`3 max`	`max`	

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.

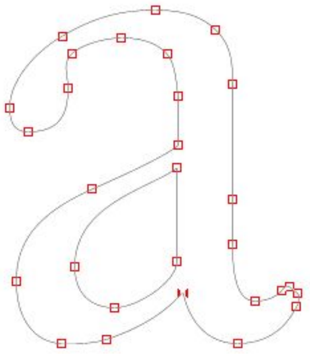
What is going on in there?

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.



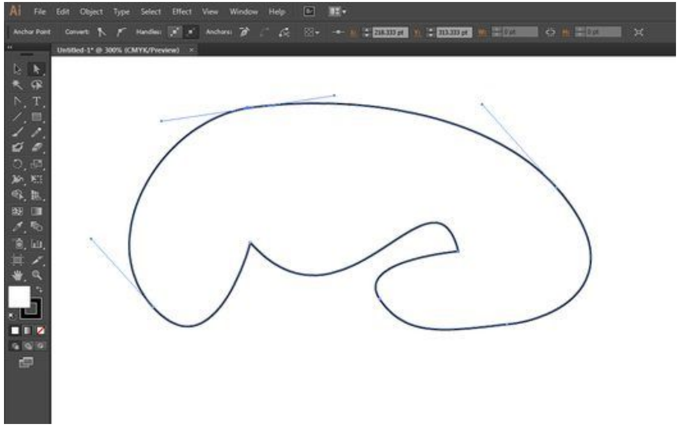
Postscript Font
(Adobe Caslon Regular)

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.



Adobe Illustrator

Adobe InDesign

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.

`% special value`
`mark`
 	
`…`
 	
`% count the values`
`% above the mark,`
`% then clear them`
`counttomark`
`cleartomark`
	`20    % 20 is on top`
	`10    % 10 is on top`
	`exch  % 10 and 20 switch`
	
	`dup   % another copy of 20`
	`pop   % one copy removed`
		`% rotate the top n values`
		`% upward k positions`
		`n k roll`
		
		`% equivalent to exch`
		`2 1 roll`

It is helpful to practice these commands with the interactive `gs` shell.

- The `pstack` command properly prints out the `mark` values.

Functions

Functions can be defined and called as follows.

`/setMediumGrey`
`{`
`  /r 0.5 def`
`  /g 0.5 def`
`  /b 0.5 def`
`  r g b setrgbcolor`
`}`
`bind def`
`setMediumGrey`
	`/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.

`10`
`{`
`  % code block`
`}`
`repeat`
	`1 2 10 % first / inc / last`
	`{`
	`  % code block`
	`}`
	`for`
		`{`
		`  …`
		`  exit`
		`  …`
		`}`
		`loop`

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, `1 2 10 {} for` completes with `1 3 5 7 9` on the stack.