CS2020
Data Structures and Algorithms
(Tutorial)

Welcome!

Today

Java Review

- Basic types, arrays, operators, etc.
- File I/O
- Static methods
- Exception handling
- Strings
- Reference passing

Document Distance Code

- Review and discussion

Primitive Data Types

Name	Size	Min	Max
byte	8 bit	-128	128
short	16 bit	-32,768	32,768
int	32 bit	-2,147,483,648	2,147,483,647
long	64 bit	-9,223,372,036,854,775,808	9,223,372,036,854,775,808
float	32 bit		
double	64 bit		
boolean	1 bit	**false**	**true**
char	16 bit (unicode)	\u0000 (0)	\uffff (65535)

Boxed Primitive Types

First principle of Java:

"Everything is an object."

Problem:

int, float, boolean, etc. are not objects.

Solution: classes for primitive types

Integer, Float, Boolean, etc.

Type Casting

Safe casting: *32 bits to 64 bits*

```
int myInteger = 7;
long myLong = (long)  myInteger;
```

Unsafe casting: *32 bits to 16 bits*

```
int myInteger = 72,325;
short myShort = (short) myInteger;
```

Arrays

Collection of elements:

```
int[] myIntegerArray;
```

Allocate memory for array:

```
myIntegerArray = new int[1000];
```

Accessing an array:

```
for (int i=0; i<1000; i++)
        myIntegerArray[i] = 7;
```

Arrays

Dynamic sized array:

```
int[] myIntegerArray;
size = getSize();
myIntegerArray = new int[size];
```

Java Operators

Operator	Functionality
=	assignment
+, –, *, /	plus, minus, multiplication, division
%	remainder
++, – –	increment, decrement
==, !=	test equality
<, >	less than, greater than
<=, >=	less-than-or-equal, greater-than-or-equal
<<, >>	left shift, right shift
&&, ||	conditional and, conditional or
~, &, ^, |	bitwise operations:complement, and, xor, or

Document Distance Main

```
package sg.edu.nus.cs2020;

public class DocumentDistanceMain
{

    public static void main(String[] args)
    {
        VectorTextFile A = new VectorTextFile("FileOne.txt");
        VectorTextFile B = new VectorTextFile("FileTwo.txt");

        double theta = VectorTextFile.Angle(A,B);

        System.out.println("The angle between A and B is: " + theta + "\n");
    }
}
```

VectorTextFile: Comments

```
/**********************************************************************
 Class: VectorTextFile
 Purpose: Represents a text file as a vector, i.e., as a sorted array of
 word/count pairs that appear in the text file.

 Constructor: VectorTextFile(String fileName)
 Behavior: Reads the specified text file and parses it appropriately.

 Public Class Methods:
    int Norm() : Returns the norm of the vector.

 Static Methods:
    double DotProduct(VectorTextFile A, VectorTextFile B) :
        Returns the dot product of two vectors.
    double Angle(VectorTextFile A, VectorTextFile B) :
        Returns the angle between two vectors.
 **********************************************************************/

// This class is part of the cs2020 package.
package sg.edu.nus.cs2020;

// This class uses the following two packages (associated with reading files):
import java.io.FileInputStream;
import java.io.IOException;
```

Member Variable Declaration

```
// Class declaration:
public class VectorTextFile {

    /*******************************
     * Class member variables
     *******************************/

    // Array of words in the file
    String[]  m_WordList;

    // Number of words in the file
    int m_FileWordCount;

    // Array of word/count pairs
    WordCountPair[] m_CountedWords;

    // Number of word/count pairs
    int m_WordPairCount;

    // Has the word list been sorted?
    boolean m_Sorted;
```

Constructor

```
//-----------------------------------------------------------------------
// Constructor: Reads and parses the specified file
//
// Input: String containing a filename
// Assumptions: fileName is a text file that exists on disk.
// Properties: On completion, m_WordList contains a sorted array of all the
// words in the text file, m_FileWordCount is the number of words in the
// text file, m_CountedWords contains a sorted array of word/count pairs
// with one entry for every distinct word in the text file, m_WordPairCount
// is the number of word/count pairs, and the flag m_Sorted is true.
// Characters in the file are treated in the following manner:
// (a) Every letter is made lower-case.
// (b) All punctuation is removed.
// (c) Each end-of-line marker ('\n') is replaced with a space.
// (d) All (other) non-letters and non-spaces are removed.
//-----------------------------------------------------------------------
public VectorTextFile(String fileName)
{
    // Begin a block of code that handles exceptions (i.e., errors)
    try{

        // First, initialize class variables
        m_WordList = null;
        m_CountedWords = null;
        m_FileWordCount = 0;
        m_WordPairCount = 0;
        m_Sorted = false;
```

Constructor

```
            // Next, read in the file and parse it into words.
            ParseFile(fileName);

            // Check for errors:
            if ((m_FileWordCount < 1) || (m_WordList == null))
            {
                throw new Exception("Reading the file failed.");
            }

            // Next, sort the words.
            InsertionSortWords();

            // Check for errors:
            if (m_Sorted == false)
            {
                throw new Exception("Sorting failed.");
            }
            VerifySort();

            // Finally, count the number of times each word appears in the file.
            CountWordFrequencies();

            // Check for errors:
            if ((m_WordPairCount < 1) || (m_CountedWords == null))
            {
                throw new Exception("Counting the word frequencies failed.");
            }
        }
        // Catch any exceptions (i.e., errors) and report problems.
        catch(Exception e)
        {
            System.out.println("Error creating VectorTextFile.");
        }
    }
```

Exception Handling

```
try{
        Do something interesting.
        Call some functions.
        Bad things might happen.
}
catch(Exception e)
{
        Deal with error.
        Recover, or re-throw exception.
}
```

Exception Handling

Some methods throw exceptions:

- int DoRiskyOperation() throws Exception

Two options:

- try/catch and handle exception
- allow exception to propagate to caller

Constructor

BAD:
Catch, but
don't handle,
all exceptions!

```
            // Next, read in the file and parse it into words.
            ParseFile(fileName);

            // Check for errors:
            if ((m_FileWordCount < 1) || (m_WordList == null))
            {
                throw new Exception("Reading the file failed.");
            }

            // Next, sort the words.
            InsertionSortWords();

            // Check for errors:
            if (m_Sorted == false)
            {
                throw new Exception("Sorting failed.");
            }
            VerifySort();

            // Finally, count the number of times each word appears in the file.
            CountWordFrequencies();

            // Check for errors:
            if ((m_WordPairCount < 1) || (m_CountedWords == null))
            {
                throw new Exception("Counting the word frequencies failed.");
            }
        }
        // Catch any exceptions (i.e., errors) and report problems.
        catch(Exception e)
        {
            System.out.println("Error creating VectorTextFile.");
        }
    }
```

Checking for errors

```
private void VerifySort() throws Exception
{
    // First, check if the sort flag is correctly set:
    if (m_Sorted == false)
    {
        throw new Exception("VerifySort fails: list not sorted.");
    }
    // Next, check if there are any words to be sorted:
    if ((m_FileWordCount < 1) || (m_WordList == null))
    {
        throw new Exception("VerifySort fails: list does not contain any words.");
    }

    // Finally, iterate through the list of words and make sure that they
    // are in properly sorted order.
    for (int i=0; i<m_FileWordCount-1; i++)
    {
        if (m_WordList[i].compareTo(m_WordList[i+1]) > 0)
        {
            throw new Exception("VerifySort fails: list badly sorted.");
        }
    }
}
```

Public Methods

```
/*****************************
 * Public Class Methods      *
 *****************************/

//---------------------------------------------------------------------
// Norm: Returns the norm of the vector.
//
// Input: None.
// Output: The norm of the vector.
// Assumptions: m_CountedWords contains a sorted list of distinct
// word/count pairs, and m_WordPairCount contains the number of word/count
// pairs.
// Methodology: The norm of a vector X is defined to be the square-root of
// DotProduct(X,X).
//---------------------------------------------------------------------
public double Norm()
{
    int dot = VectorTextFile.DotProduct(this, this);
    return Math.sqrt(dot);
}
```

Static Methods

```
static int DotProduct(VectorTextFile A, VectorTextFile B)
{
    // Initialize local variables:

    // The sum is initially zero
    int sum = 0;

    // Alength is the number of word/count pairs in A
    int Alength = A.m_WordPairCount;
    // We begin with word/count pair zero
    int Aindex = 0;

    // Blength is the number of word/count pairs in B
    int Blength = B.m_WordPairCount;
    // We begin with word/count pair zero
    int Bindex = 0;

    // We iterate through all the word/count pairs, looking for words that
    // appear in both A and B.  We continue until we run out of words in
    // either A or B.
    while ((Aindex < Alength) && (Bindex < Blength))
    {
        // First, get the word associated with Aindex in A
        WordCountPair Awordpair = A.m_CountedWords[Aindex];
        String Aword = Awordpair.getWord();

        // Next, get the word associated with Bindex in B
        WordCountPair Bwordpair = B.m_CountedWords[Bindex];
        String Bword = Bwordpair.getWord();
```

Static Methods / Variables

Special relationship:

- Part of a class, not an object.
- No access to object member variables.

Usage:

ClassName.*staticfunction*(...)

Bad usage:

ClassName obj;

obj.*staticfunction*(...)

Static Methods

```
static int DotProduct(VectorTextFile A, VectorTextFile B)
{
    // Initialize local variables:

    // The sum is initially zero
    int sum = 0;

    // Alength is the number of word/count pairs in A
    int Alength = A.m_WordPairCount;
    // We begin with word/count pair zero
    int Aindex = 0;

    // Blength is the number of word/count pairs in B
    int Blength = B.m_WordPairCount;
    // We begin with word/count pair zero
    int Bindex = 0;

    // We iterate through all the word/count pairs, looking for words that
    // appear in both A and B.  We continue until we run out of words in
    // either A or B.
    while ((Aindex < Alength) && (Bindex < Blength))
    {
        // First, get the word associated with Aindex in A
        WordCountPair Awordpair = A.m_CountedWords[Aindex];
        String Aword = Awordpair.getWord();

        // Next, get the word associated with Bindex in B
        WordCountPair Bwordpair = B.m_CountedWords[Bindex];
        String Bword = Bwordpair.getWord();
```

Static Methods

```
        // If Aword==Bword, then we have found a word in both vector A and B
        if (Aword.equals(Bword))
        {
            // Add the product of the counts to the sum.
            sum += (Awordpair.getCount()*Bwordpair.getCount());

            // Next, increment both Aindex and Bincex to consider the next
            // word in both vectors.
            Aindex++;
            Bindex++;
        }
        else if (Aword.compareTo(Bword) > 0)
        {
            // Otherwise, if (Aword > Bword) in alphabetic order, we
            // increment Bindex, going to the next WordCountPair in B.
            Bindex++;
        }
        else
        {
            // Otherwise, (Bword > Aword) in alphabetic order.  We
            // increment Aindex, going to the next WordCountPair in A.
            Aindex++;
        }
    }

    // Finally, we return the dot-product.
    return sum;
}
```

Angle

```
static double Angle(VectorTextFile A, VectorTextFile B)
{
    // First calculate the dot product of the vectors A and B
    int dot = VectorTextFile.DotProduct(A, B);

    // Second, calculate the norm of the two vectors
    double Anorm = A.Norm();
    double Bnorm = B.Norm();

    // Third, calculate (AB)/(|A|*|B|)
    double result = dot/(Anorm*Bnorm);

    // Lastly, take the arccos of the result
    double theta = Math.acos(result);

    // Return the angle
    return theta;
}
```

ParseFile

```
private void ParseFile(String fileName) throws IOException
{
    // First, read the file into a single long string.
    String strTextFile = ReadFile(fileName);

    // Next, divide the string into words.
    SplitString(strTextFile);
}
```

ReadFile

```
private String ReadFile(String fileName) throws IOException
{
    // Declare and initialize variables
    FileInputStream inputStream = null;
    String strTextFile = "";
    int iSize = 0;

    // Begin a block of code that handles exceptions
    try{
        // Open the file as a stream
        inputStream = new FileInputStream(fileName);

        // Determine the size of the file, in bytes
        iSize = inputStream.available();

        // Read in the file, one character at a time.
        // For each character, normalize it, removing punctuation and capitalization.
        for (int i=0; i<iSize; i++)
        {
            // Read a character
            char c = (char)inputStream.read();

            // Ensure that the character is lowercase
            c = Character.toLowerCase(c);
```

Reading and Writing Files

- Many different *stream* types:
  - Byte Streams: raw data
  - Character Streams: character data (unicode, etc.)
  - Buffered Streams
  - Scanner: reads tokens
  - Data Streams
  - Object Streams

Byte Streams

Two classes:

- java.io.FileInputStream
- java.io.FileOutputStream

Note:

import java.io.FileInputStream

import java.io.IOException

etc.

Byte Streams

FileInputStream:

Constructor: FileInputStream(filename)

read()

close()

available()

skip(long n)

- Exceptions:

  FileNotFoundException

Byte Streams

FileOutputStream:

- Constructor: FileOutputStream(filename)
- write(byte[] b)
- write(int b)
- close()

Other Streams

Character streams:

- java.io.FileReader
- java.io.FileWriter

Buffer streams (wrap other streams):

- java.io.BufferedInputStream
- java.io.BufferedOutputStream
- java.io.BufferedReader
- java.io.BufferedWriter

Standard Streams

Pre-defined streams:

- System.in
- System.out
- System.err

Example:

```
System.out.println(“Here is some text.”);
```

Object Streams

- Write an entire object to a file:
  - java.io.ObjectInputStream
  - java.io.ObjectOutputStream
- Example:

```
myObject obj;
FileOutputStream outStr = new FileOutputStream(“filename”);
ObjectOutputStream objStr = new ObjectOutputStream(outStr);
objStr.writeObject(obj);
```

ReadFile

```
private String ReadFile(String fileName) throws IOException
{
    // Declare and initialize variables
    FileInputStream inputStream = null;
    String strTextFile = "";
    int iSize = 0;

    // Begin a block of code that handles exceptions
    try{
        // Open the file as a stream
        inputStream = new FileInputStream(fileName);

        // Determine the size of the file, in bytes
        iSize = inputStream.available();

        // Read in the file, one character at a time.
        // For each character, normalize it, removing punctuation and capitalization.
        for (int i=0; i<iSize; i++)
        {
            // Read a character
            char c = (char)inputStream.read();

            // Ensure that the character is lowercase
            c = Character.toLowerCase(c);
```

```
            // Check if the character is a letter
            if (Character.isLetter(c))
            {
                strTextFile = strTextFile + c;
            }
            // Check if the character is a space or an end-of-line marker
            else if ((c == ' ') || (c == '\n'))
            {
                // In this case, we add a space to the string.
                // Note: only add a space if the previous character
                // is not also a space.  This prevents adding two spaces in a row.
                if (!strTextFile.endsWith(" "))
                {
                        // Add a space:
                        strTextFile = strTextFile + ' ';
                }
            }
            else
            {
                // Do nothing: skip this character.
            }
        }
    }   // end of try block
    finally // handle any exceptions
    {
        // If the file is open, then close it.
        if (inputStream != null)
        {
            inputStream.close();
        }
    }

    // Return the string representing the text file
    return strTextFile;
}
```

```
// Begin a block of code that handles exceptions
try{
    // Open the file as a stream
    inputStream = new FileInputStream(fileName);

    // Determine the size of the file, in bytes
    iSize = inputStream.available();

    // Initialize the char buffer to be arrays of the appropriate size.
    charBuffer = new char[iSize];

    // Read in the file, one character at a time.
    // For each character, normalize it, removing punctuation and capitalization.
    for (int i=0; i<iSize; i++)
    {
        // Read a character
        char c = (char)inputStream.read();

        // Ensure that the character is lower-case
        c = Character.toLowerCase(c);

        // Check if the character is a letter, or whitespace, or a new line
        if (Character.isLetter(c))
        {
            charBuffer[iCharCount] = c;
            iCharCount++;
        }
        else if ((c == ' ') || (c == '\n'))
        {
```

Character Class

Wrapper class for char:

java.lang.Character

Static methods:

- isDigit(char ch)
- isWhitespace(char ch)
- isLetter(char ch)
- getNumericValue(char ch)
- toUpperCase(char ch)
- toString(char ch)
- etc.

SplitString

```
private void SplitString(String strTextFile)
{
    // Initialize local variables:
    int iStringSize = strTextFile.length();
    int iWordCount = 0;
    String word = "";

    // Initialize class variables:
    // Note: the length of the string is an overestimate of the number of words in the String.
    // As a result, we allocate too much space, wasting memory.
    // It would be better to allocate space more efficiently.
    m_WordList = new String[iStringSize];

    // Iterate through the string, examining every character.
    // Accumulate characters in words, detecting word breaks.
    for (int i=0; i<iStringSize; i++)
    {
        // Read a character
        char c = strTextFile.charAt(i);

        // Check if the character is part of a word, or a word break.
        if (c != ' ')
        {
            // Here, the character is part of a word, so add it to the word.
            word += c;
        }
        else
        {
```

SplitString

```
            else
            {
                // Otherwise, the character is not part of a word.
                // If we have found a non-empty word, add it to the list of words.
                if (word != "")
                {
                    // Add the word to the list of words
                    m_WordList[iWordCount] = word;

                    // Increment the number of words discovered.
                    iWordCount++;

                    // Reinitialize the word to the empty string
                    word = "";
                }
            }
        }

        // Check if there is any leftover word, once the string is complete.
        // If so, add the word to the word list, and increment the word count.
        if (word != "")
        {
            m_WordList[iWordCount] = word;
            iWordCount++;
        }

        // Save the word count.
        m_FileWordCount = iWordCount;
    }
```

String

Creating strings:

```
String str = “Some text.”;
String altstr = new String(“some text”);
```

Accessing a string:

- charAt(int index)
- substring(int begin, int end)
- toCharArray()
- length()

String

Comparing strings:

- compareTo(String otherString)
- compareToIgnoreCase(String otherString)
- equals(Object anObject)

Using strings:

- Flexible and easy: str = str + ‘c’;
- Use with care...

CountWordFrequencies

```
private void CountWordFrequencies() throws Exception
{
    // Check for errors:
    if ((m_WordList==null) || (m_FileWordCount<1) || (m_Sorted == false))
    {
        throw new Exception("Failed in CountWordFrequencies: no words to count.");
    }

    // Initialize the m_CountedWords array.
    // We use m_FileWordCount as a safe estimate of the number of distinct
    // words in m_WordList.  Notice that this is an inefficient use of space,
    // as m_CountedWords will likely be much smaller than m_WordList.
    m_CountedWords = new WordCountPair[m_FileWordCount];

    // Initialize the number of count/value pairs to zero.
    int iNumPairs = 0;

    // Initialize the first word to be the first word in the word list.
    String word = m_WordList[0];
    // Initialize the count to be one.
    int count = 1;
```

CountWordFrequencies

```
    // Iterate through every word in m_WordList
    for (int i=1; i<m_FileWordCount; i++)
    {
        // If we find another copy of the word:
        if (m_WordList[i].equals(word))
        {
            // Then increment the count.
            count++;
        }
        else
        {
            // Otherwise, we have found a new word.
            // Store the old word and its count as new WordCountPair.
            m_CountedWords[iNumPairs] = new WordCountPair(word, count);
            // Increment the number of word/count pairs that we have discovered.
            iNumPairs++;

            // Re-initialize word with the newly found word.
            word = m_WordList[i];
            // Re-initialize the count to be 1.
            count = 1;
        }
    }
    // Save the number of word/count pairs in m_WordPairCount
    m_WordPairCount = iNumPairs;
}
```

InsertionSort

```
private void InsertionSortWords() throws Exception
{
    // Initialize local variables:

    // index stores the slot in the array that we are trying to fill
    int index = 0;
    // strMin stores the word we are currently sorting into place
    String SortString = null;
    // iMax stores the index of the largest sorted word
    int iMaxSorted = 0;

    // Check for errors
    if ((m_WordList==null) || (m_FileWordCount==0))
    {
        throw new Exception("Failed in InsertionSortWords: no words to sort.");
    }
```

InsertionSort

```
    for (iMaxSorted = 0; iMaxSorted<m_FileWordCount-1; iMaxSorted++)
    {
        // First, fix the string we are going to sort into place
        SortString = m_WordList[iMaxSorted+1];

        // We need to find where SortString fits in the array [1..iMaxSorted+1]
        index = iMaxSorted+1;
        while (index > 0 && SortString.compareTo(m_WordList[index-1]) < 0)
        {
            m_WordList[index] = m_WordList[index-1];
            index--;
        }

        // Now that we have found where SortString goes,
        // move it into place.
        m_WordList[index] = SortString;
    }
    // Now that we are done sorting, set a flag indicating that the sort is complete.
    m_Sorted = true;
}
```

```
private void MergeSortWords(int Begin, int End) throws Exception
{
    // First, check for errors
    if (End < Begin)
    {
        throw new Exception("Failed MergeSortWords: End is not greater than Begin.");
    }
    if ((m_WordList==null) || (m_FileWordCount<1))
    {
        throw new Exception("Failed in MergeSortWords: no words to sort.");
    }

    // Determine the number of words in the array to sort
    int NumWords = End-Begin+1;

    // If there is only one element in the list to sort, then
    // by definition, it is already well sorted.
    if (NumWords < 2)
    {
        return;
    }

    // We now divide the list into two parts, each 1/2 the size
    // of the initial list.  The first list is from [Begin..Middle-1]
    // and the second list is from [Middle..End].
    //
    // Note that division by two automatically rounds to an integer.
    int Middle = Begin + NumWords/2;

    // Recursively sort each half-list.
    MergeSortWords(Begin, Middle-1);
    MergeSortWords(Middle, End);


    // Merge the two sorted lists.
    Merge(Begin, Middle, End);
}
```

MergeSort

```
private void MergeSortWords(int Begin, int End) throws Exception
{
    // First, check for errors
    if (End < Begin)
    {
        throw new Exception("Failed MergeSortWords: End is not greater than Begin.");
    }
    if ((m_WordList==null) || (m_FileWordCount<1))
    {
        throw new Exception("Failed in MergeSortWords: no words to sort.");
    }

    // Determine the number of words in the array to sort
    int NumWords = End-Begin+1;

    // If there is only one element in the list to sort, then
    // by definition, it is already well sorted.
    if (NumWords < 2)
    {
        return;
    }
```

MergeSort

```
    // We now divide the list into two parts, each 1/2 the size
    // of the initial list.  The first list is from [Begin..Middle-1]
    // and the second list is from [Middle..End].
    //
    // Note that division by two automatically rounds to an integer.
    int Middle = Begin + NumWords/2;

    // Recursively sort each half-list.
    MergeSortWords(Begin, Middle-1);
    MergeSortWords(Middle, End);


    // Merge the two sorted lists.
    Merge(Begin, Middle, End);
}
```

Reference Passing

Example:


```
void BadSwap(Object a, Object b)
{
        Object temp = a
        a = b
        b = temp
        return;
}
```

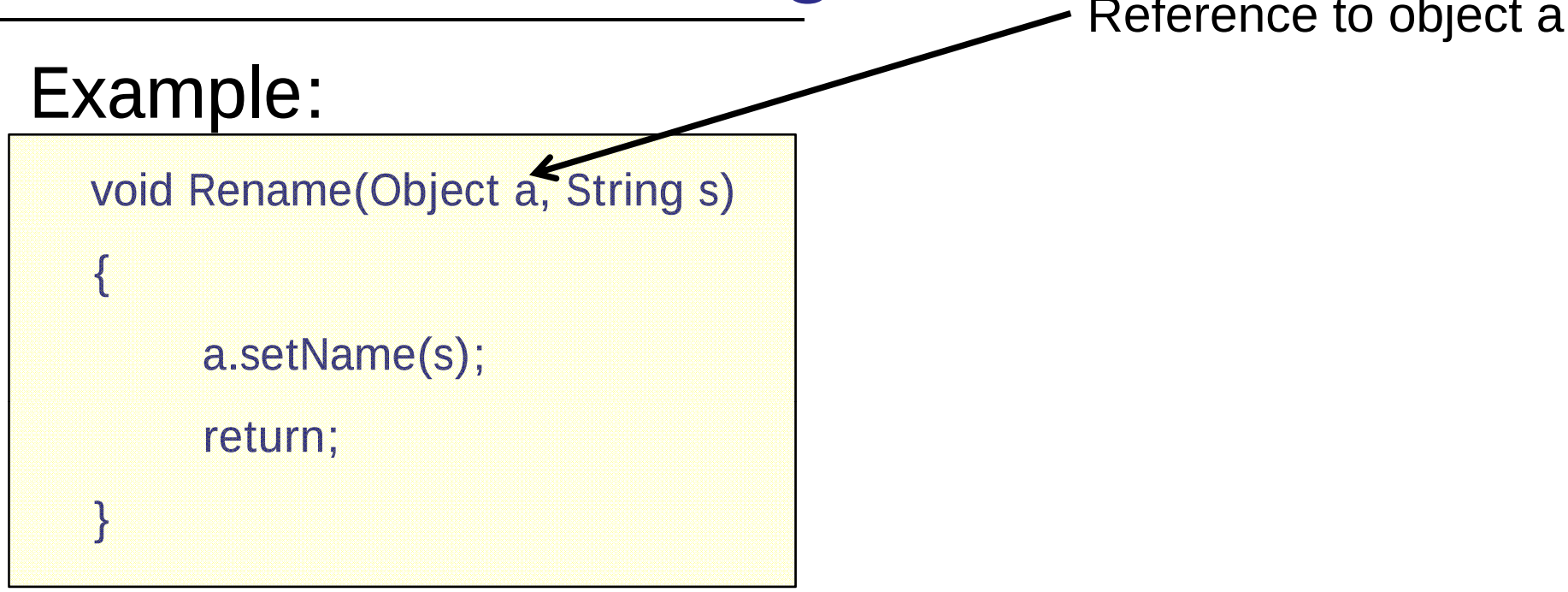
Then:

```
Object Bob = new("bob");
Object Joe = new("joe");
Test(Bob, Joe)
```

No change to Bob or Joe!

Reference Passing

Example:



Then:

```
Object Bob = new("bob");
Rename(Bob, "Joe")
```

Bob renamed to Joe!