



COLLECTIONS AND FILES

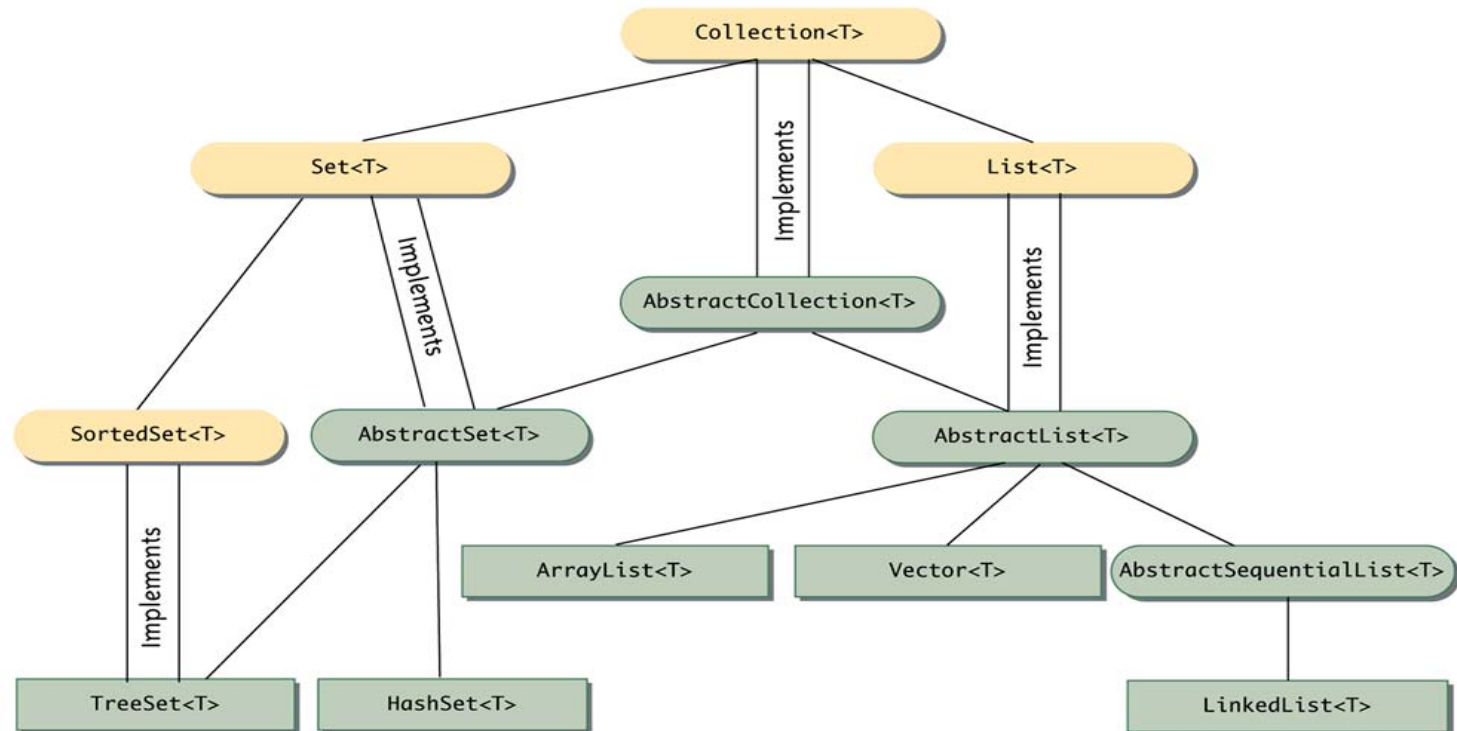
CIS*2430 (Fall 2010)

Java Collections

- A Java collection is any class that holds objects and implements the **Collection** interface
 - Example: the **ArrayList<T>** class
- A collection can be accessed by an iterator, which is an object that cycles through all the elements in the collection
- All the collection classes can be found in package `java.util`

The Collection Landscape

Display 16.1 The Collection Landscape



Interface

Abstract Class

Concrete Class

A single line between two boxes means the lower class or interface is derived from (extends) the higher one.

T is a type parameter for the type of the elements stored in the collection.

Collection Relationships

- Classes that implement the `List<T>` interface have their elements ordered as on a list
 - Elements are indexed starting from zero
 - A class that implements the `List<T>` interface allows elements to occur more than once
 - The `List<T>` interface has more methods than the `Collection<T>` interface
 - The `ArrayList<T>` class implements the `List<T>` interface

Iterators

- An iterator is an object used with a collection to provide sequential access to the collection elements
 - This access allows examination and possible modification of the elements
- An iterator imposes an ordering on the elements of a collection even if the collection itself does not impose any order on its elements
 - If the collection does impose an ordering on its elements, then the iterator will use the same ordering

Iterator<T> Interface

- Java provides an **Iterator<T>** interface
 - Any object of any class that satisfies the **Iterator<T>** interface is an **Iterator<T>**
- An **Iterator<T>** does not stand on its own
 - It must be associated with some collection object using the method **iterator**
 - If **c** is an instance of a collection class (e.g., **HashSet<String>**), the following obtains an iterator for **c**:
Iterator iteratorForC = c.iterator();

Iterator Example (1 / 3)

Display 16.8 An Iterator

```
1  import java.util.HashSet;
2  import java.util.Iterator;

3  public class HashSetIteratorDemo
4  {
5      public static void main(String[] args)
6      {
7          HashSet<String> s = new HashSet<String>();

8          s.add("health");
9          s.add("love");
10         s.add("money");

11         System.out.println("The set contains:");
```

(continued)

Iterator Example (2/3)

Display 16.8 An Iterator

```
12      Iterator<String> i = s.iterator();
13      while (i.hasNext())
14          System.out.println(i.next());

15      i.remove();

16      System.out.println();
17      System.out.println("The set now contains:");

18      i = s.iterator();
19      while (i.hasNext())
20          System.out.println(i.next());

21      System.out.println("End of program.");
22  }
23 }
```

You cannot “reset” an iterator “to the beginning.” To do a second iteration, you create another iterator.

(continued)

Iterator Example (3/3)

Display 16.8 An Iterator

SAMPLE DIALOGUE

The set contains:

money

love

health

The set now contains:

money

love

End of program.

The `HashSet<T>` object does not order the elements it contains, but the iterator imposes an order on the elements.

For-Each Loops as Iterators (1 / 2)

Display 16.9 For-Each Loops as Iterators

```
1  import java.util.HashSet;
2  import java.util.Iterator;

3  public class ForEachDemo
4  {
5      public static void main(String[] args)
6      {
7          HashSet<String> s = new HashSet<String>();

8          s.add("health");
9          s.add("love");
10         s.add("money");

11         System.out.println("The set contains:");
```

(continued)

For-Each Loops as Iterators (2/2)

Display 16.9 For-Each Loops as Iterators

```
12      String last = null;
13      for (String e : s)
14      {
15          last = e;
16          System.out.println(e);
17      }

18      s.remove(last);

19      System.out.println();
20      System.out.println("The set now contains:");

21      for (String e : s)
22          System.out.println(e);

23      System.out.println("End of program.");
24  }
25 }
```

The output is the same as in Display 16.8.

Iterators for ArrayLists (1 / 4)

Display 16.12 An Iterator Returns a Reference

```
1  import java.util.ArrayList;
2  import java.util.Iterator;

3  public class IteratorReferenceDemo
4  {
5      public static void main(String[] args)
6      {
7          ArrayList<Date> birthdays = new ArrayList<Date>();

8          birthdays.add(new Date(1, 1, 1990));
9          birthdays.add(new Date(2, 2, 1990));
10         birthdays.add(new Date(3, 3, 1990));

11         System.out.println("The list contains:");
```

*The class **Date** is defined in Display 4.13, but you can easily guess all you need to know about **Date** for this example.*

(continued)

Iterators for ArrayLists (2/4)

Display 16.12 An Iterator Returns a Reference

```
12      Iterator<Date> i = birthdays.iterator();
13      while (i.hasNext())
14          System.out.println(i.next());

15      i = birthdays.iterator();
16      Date d = null; //To keep the compiler happy.
17      System.out.println("Changing the references.");
18      while (i.hasNext())
19      {
20          d = i.next();
21          d.setDate(4, 1, 1990);
22      }
```

(continued)

Iterators for ArrayLists (3/4)

Display 16.12 An Iterator Returns a Reference

```
23      System.out.println("The list now contains:");

24      i = birthdays.iterator();
25      while (i.hasNext())
26          System.out.println(i.next());

27      System.out.println("April fool!");
28  }
29 }
```

(continued)

Iterators for ArrayLists (4/4)

Display 16.12 An Iterator Returns a Reference

SAMPLE DIALOGUE

The list contains:

January 1, 1990

February 2, 1990

March 3, 1990

Changing the references.

The list now contains:

April 1, 1990

April 1, 1990

April 1, 1990

April fool!

Maps

- The Java *map* framework deals with collections of ordered pairs
 - Ordered pair: a key and an associated value
- Objects in the map framework can implement mathematical functions and relations, so can be used to construct database classes
- The map framework uses the `Map<K, V>` interface and the two concrete classes are `TreeMap<K, V>` and `HashMap<K, V>`

HashMap Example (1 / 3)

```
import java.util.HashMap;
import java.util.Scanner;

public class HashMapDemo {
    public static void main(String[] args) {
        HashMap<String, Employee> employees =
            new HashMap<String, Employee>(10);

        employees.put("Joe",
            new Employee("Joe", new Date("September", 15, 1970)));
        employees.put("Andy",
            new Employee("Andy", new Date("August", 22, 1971)));
        employees.put("Greg",
            new Employee("Greg", new Date("March", 9, 1972)));
        employees.put("Kiki",
            new Employee("Kiki", new Date("October", 8, 1970)));
        System.out.println("Added Joe, Andy, Greg, and Kiki to the map.");
    }
}
```

HashMap Example (2/3)

```
Scanner keyboard = new Scanner(System.in);
String name = "";
do {
    System.out.print("\nEnter a name to look up the map. ");
    System.out.println("Press enter to quit.");
    name = keyboard.nextLine();
    if (employees.containsKey(name)) {
        Employee e = employees.get(name);
        System.out.println("Name found: " + e.toString());
    } else if (!name.equals("")) {
        System.out.println("Name not found");
    }
} while (!name.equals(""));
}
```

HashMap Example (3/3)

Sample Dialogue:

Added Joe, Andy, Greg, and Kiki to the map.

Enter a name to look up in the map. Press enter to quit.

Joe

Name found: Joe September 15, 1970

Enter a name to look up in the map. Press enter to quit.

Kiki

Name found: Kiki October 8, 1970

Enter a name to look up in the map. Press enter to quit.

Myla

Name not found.

Indices and Search

- Given the title “java programming”, we can extract two tokens “java” and “programming”.
- If “java” appears in records 0, 3, 7, and 10, we can add the following entry to a HashMap index:
key: “java”
value: a list of [0, 3, 7, 10]
- If we further know that “programming” appears in records [0, 5, 7, 12, 15], then the search for “java programming” will be the intersection of [0, 3, 7, 10] and [0, 5, 7, 12, 15], which is equal to [0, 7].

File Input/Output

- A *stream* is an object that enables the flow of data between a program and some I/O device or file
- Input streams can flow from the keyboard or from a file
 - `System.in` is an input stream that connects to the keyboard
`Scanner keyboard = new Scanner(System.in);`
- Output streams can flow to a screen or to a file
 - `System.out` is an output stream that connects to the screen
`System.out.println("Output stream");`

Text Files

- Files that are designed to be read by human beings, and that can be read or written with an editor are called *text files*
 - Text files are also called ASCII files because the data they contain uses an ASCII encoding scheme
 - An advantage of text files is that they are usually the same on all computers, and as a result can be moved from one computer to another

Binary Files

- Files that are designed to be read by programs and that consist of a sequence of binary digits are called *binary files*
 - Binary files are designed to be read on the same type of computer and with the same programming language as the computer that created the file
 - An advantage of binary files is that they are *more efficient to process* than text files
 - Unlike most binary files, Java binary files have the advantage of being platform independent also

Writing to a Text File

- The class `PrintWriter` is a stream class that can be used to write to a text file
- All file I/O classes are in the package `java.io` and need to be imported:

```
import java.io.PrintWriter;  
import java.io.FileOutputStream;  
import java.io.FileNotFoundException;
```
- The class `PrintWriter` has no constructor that takes a file name as its argument
 - It uses another class, `FileOutputStream`, to convert a file name to an object that can be used as the argument to its constructor

Writing to a Text File

- The process of connecting a stream to a file is called *opening the file*

```
PrintWriter outputStream =  
    new PrintWriter(new FileOutputStream(FileName));
```

- If the file already exists, then doing this causes the old contents to be lost
- If the file does not exist, then a new, empty file named *FileName* is created

Writing Output File

```
import java.io.PrintWriter;
import java.io.FileOutputStream;
import java.io.FileNotFoundException;

public class TextFileOutputDemo {
    public static void main(String[] args) {
        PrintWriter outputStream = null;
        try {
            outputStream = new PrintWriter(new FileOutputStream("stuff.txt"));
        } catch (FileNotFoundException e) {
            System.out.println("Error opening the file stuff.txt.");
            System.exit(0);
        }
        outputStream.println("The quick brown fox");
        outputStream.println("jumped over the lazy dog.");
        outputStream.close();
    }
}
```

Buffered Output

- Output streams connected to files are usually *buffered*
 - Rather than physically writing to the file as soon as possible, the data are saved in a temporary memory location (*buffer*)
 - When enough data accumulates, or when the method **flush** is invoked, the buffered data are written to the file all at once
 - This is more efficient, since physical writes to a file can be slow

Closing a File

- The method **close** invokes the method **flush**, thus insuring that all the data are written to the file
 - Although Java closes a file automatically when a program ends, it is safer to close it explicitly
 - If a program relies on Java to close the file, and the program terminates abnormally, then any output that was buffered may not get written to the file
 - Also, if a program writes to a file and later reopens it to read from the same file, it will have to be closed first anyway

Pitfall: a `try` Block is a Block

- Since opening a file can result in an exception, it should be placed inside a `try` block
- If the variable for a `PrintWriter` object needs to be used outside that block, then the variable must be declared outside the block
 - Otherwise it would be local to the block, and could not be used elsewhere
 - If it were declared in the block and referenced elsewhere, the compiler will generate a message indicating that it is an undefined identifier

Appending to a Text File

- To create a **PrintWriter** object and connect it to a text file for *appending*, a second argument can be used in the constructor for the **FileOutputStream** object

```
outputStreamName = new PrintWriter(new  
    FileOutputStream(FileName, true));
```

- After this statement, the methods **print**, **println** and/or **printf** can be used to write to the file
- The new text will be written *after the old text* in the file

Reading from a Text File

- The class **Scanner** can be used for reading from the keyboard as well as reading from a text file

```
Scanner StreamObject =  
    new Scanner(new FileInputStream(fileName));
```

- Methods of the **Scanner** class for reading input behave the same whether reading from the keyboard or reading from a text file
 - For example, the **nextInt** and **nextLine** methods

Reading Input File (1 / 4)

Display 10.3 Reading Input from a Text File Using Scanner

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4
5  public class TextFileScannerDemo
6  {
7      public static void main(String[] args)
8      {
9          System.out.println("I will read three numbers and a line");
10         System.out.println("of text from the file morestuff.txt.");
11
12         Scanner inputStream = null;
13
14         try
15         {
16             inputStream =
17                 new Scanner(new FileInputStream("morestuff.txt"));
18         }
```

(continued)

Reading Input File (2/4)

Display 10.3 Reading Input from a Text File Using Scanner

```
19      catch(FileNotFoundException e)
20      {
21          System.out.println("File morestuff.txt was not found");
22          System.out.println("or could not be opened.");
23          System.exit(0);
24      }
25      int n1 = inputStream.nextInt( );
26      int n2 = inputStream.nextInt( );
27      int n3 = inputStream.nextInt( );
28
29      inputStream.nextLine(); //To go to the next line
30
31      String line = inputStream.nextLine( );
32
```

(continued)

Reading Input File (3/4)

Display 10.3 Reading Input from a Text File Using Scanner

```
33      System.out.println("The three numbers read from the file are:");
34      System.out.println(n1 + ", " + n2 + ", and " + n3);
35
36      System.out.println("The line read from the file is:");
37      System.out.println(line);
38
39      inputStream.close( );
40  }
41 }
```

File morestuff.txt

```
1 2
3 4
Eat my shorts.
```

This file could have been made with a text editor or by another Java program.

(continued)

Reading Input File (4/4)

Display 10.3 Reading Input from a Text File Using Scanner

SCREEN OUTPUT

```
I will read three numbers and a line  
of text from the file morestuff.txt.  
The three numbers read from the file are:  
1, 2, and 3  
The line read from the file is:  
Eat my shorts.
```

Test for End of File

- A program that tries to read beyond the end of a file using methods of the **Scanner** class will cause an exception to be thrown
- However, instead of having to rely on an exception to signal the end of a file, the **Scanner** class provides methods such as **hasNextInt** and **hasNextLine**
 - These methods can also be used to check that the next token to be input is a suitable element of the appropriate type

Testing with hasNextInt (1 / 2)

Display 10.5 Checking for the End of a Text File with hasNextInt

```
1  import java.util.Scanner;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;

4  public class HasNextIntDemo
5  {
6      public static void main(String[] args)
7      {
8          Scanner inputStream = null;

9          try
10         {
11             inputStream =
12                 new Scanner(new FileInputStream("data.txt"));
13         }
14         catch(FileNotFoundException e)
15         {
16             System.out.println("File data.txt was not found");
17             System.out.println("or could not be opened.");
18             System.exit(0);
19         }
```

(continued)

Testing with hasNextInt (2/2)

Display 10.5 Checking for the End of a Text File with hasNextInt

```
20     int next, sum = 0;
21     while (inputStream.hasNextInt( ))
22     {
23         next = inputStream.nextInt( );
24         sum = sum + next;
25     }

26     inputStream.close( );

27     System.out.println("The sum of the numbers is " + sum);
28 }
29 }
```

File data.txt

```
1  2
3  4 hi 5
```

*Reading ends when either the end of the file is reached or a token that is not an **int** is reached. So, the 5 is never read.*

SCREEN OUTPUT

The sum of the numbers is 10

Path Names

- A *path name* not only gives the name of the file, but also the directory or folder in which the file exists
- A *full path name* gives a complete path name, starting from the root directory
- A *relative path name* gives the path to the file, starting with the directory in which the program is located

Path Names

- The way path names are specified depends on the operating system
 - A typical UNIX path name that could be used as a file name argument is

`"/user/joe/data/data.txt"`

- A **Scanner** input stream connected to this file is created as follows:

```
Scanner inputStream =  
    new Scanner(new  
        FileInputStream("/user/joe/data/data.txt"));
```


Path Names

- The Windows operating system specifies path names in a different way

- A typical Windows path name is the following:

`C:\dataFiles\goodData\data.txt`

- A `Scanner` input stream connected to this file is created as follows:

```
Scanner inputStream = new  
    Scanner(new FileInputStream  
        ("C:\\dataFiles\\goodData\\data.txt"));
```

Path Names

- A double backslash (`\\`) must be used for a Windows path name enclosed in a quoted string
 - This problem does not occur with path names read in from the keyboard
- Problems with escape characters can be avoided altogether by always using UNIX conventions when writing a path name
 - A Java program will accept a path name written in either Windows or Unix format regardless of the operating system on which it is run

The File Class

- The **File** class is like a wrapper class for file names
 - The constructor for the class **File** takes a name, (known as the *abstract name*) as a string argument, and produces an object that represents the file with that name
 - The **File** object and methods of the class **File** can be used to determine information about the file and its properties

Methods in File Class (1 / 5)

Display 10.12 Some Methods in the Class File

File is in the `java.io` package.

```
public File(String File_Name)
```

Constructor. *File_Name* can be either a full or a relative path name (which includes the case of a simple file name). *File_Name* is referred to as the **abstract path name**.

```
public boolean exists()
```

Tests whether there is a file with the abstract path name.

```
public boolean canRead()
```

Tests whether the program can read from the file. Returns `true` if the file named by the abstract path name exists and is readable by the program; otherwise returns `false`.

(continued)

Methods in File Class (2/5)

Display 10.12 Some Methods in the Class File

```
public boolean setReadOnly()
```

Sets the file represented by the abstract path name to be read only. Returns `true` if successful; otherwise returns `false`.

```
public boolean canWrite()
```

Tests whether the program can write to the file. Returns `true` if the file named by the abstract path name exists and is writable by the program; otherwise returns `false`.

```
public boolean delete()
```

Tries to delete the file or directory named by the abstract path name. A directory must be empty to be removed. Returns `true` if it was able to delete the file or directory. Returns `false` if it was unable to delete the file or directory.

(continued)

Methods in File Class (3/5)

Display 10.12 Some Methods in the Class File

```
public boolean createNewFile() throws IOException
```

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns true if successful, and returns false otherwise.

```
public String getName()
```

Returns the last name in the abstract path name (that is, the simple file name). Returns the empty string if the abstract path name is the empty string.

```
public String getPath()
```

Returns the abstract path name as a String value.

```
public boolean renameTo(File New_Name)
```

Renames the file represented by the abstract path name to *New_Name*. Returns true if successful; otherwise returns false. *New_Name* can be a relative or absolute path name. This may require moving the file. Whether or not the file can be moved is system dependent.

(continued)

Methods in File Class (4/5)

Display 10.12 Some Methods in the Class File

```
public boolean isFile()
```

Returns true if a file exists that is named by the abstract path name and the file is a normal file; otherwise returns false. The meaning of *normal* is system dependent. Any file created by a Java program is guaranteed to be normal.

```
public boolean isDirectory()
```

Returns true if a directory (folder) exists that is named by the abstract path name; otherwise returns false.

(continued)

Methods in File Class (5/5)

Display 10.12 Some Methods in the Class File

```
public boolean mkdir()
```

Makes a directory named by the abstract path name. Will not create parent directories. See `makedirs`. Returns `true` if successful; otherwise returns `false`.

```
public boolean mkdirs()
```

Makes a directory named by the abstract path name. Will create any necessary but nonexistent parent directories. Returns `true` if successful; otherwise returns `false`. Note that if it fails, then some of the parent directories may have been created.

```
public long length()
```

Returns the length in bytes of the file named by the abstract path name. If the file does not exist or the abstract path name names a directory, then the value returned is not specified and may be anything.