More I/O, Collections, Compression

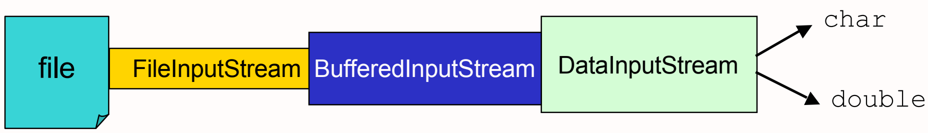
Sara Sprenkle

June 22, 2005

Review

- I/O: Streams
  - Character, Byte
- Files
- Assignment 2 due

A More Connected Stream



- FileInputStream reads bytes from the file
- BufferedInputStream buffers bytes
  - speeds up access to the file.
- DataInputStream reads buffered bytes as types

FYI: Additional I/O Functionality

- Java provides classes so that you can
  - Lock files (`java.nio.channels.FileLock`)
    - Coordinates accesses to files
      - Multiple programs read/write same file
    - Depends on OS to enforce locks
  - Read from random points in the file
    - `java.io.RandomAccessFile`

Parsing Files

- Use programs to automate tasks
- Often have large amounts of data in files
- Java provides classes to make parsing easier

StringTokenizer

- Lexical analyzer
  - Parse text
- Breaks a string into tokens
- Example:

```
StringTokenizer st = new StringTokenizer("this is a test");
while (st.hasMoreTokens()) {
     System.out.println(st.nextToken());
}
```

Output: this
is
a
test

String Tokenizer

- Optional constructor: define a delimiter
  - Default delimiter: `" \t\n\r\f"`
    - The first character is a space
  - Used to separate tokens
  - Delimiters do not count as tokens
  - How could you parse a CSV file?
- Legacy class
  - Maintained for backwards compatibility

Alternative: use String class

Regular expression:
\\s means whitespace

```
String test = "this is a test";
String[] result = test.split("\\s");
for (int x=0; x<result.length; x++)
    System.out.println(result[x]);
```

Output: this
is
a
test

FYI: StreamTokenizer

- Tokenize an incoming character stream
- Table-driven lexical analyzer
  - every possible input character has a significance
  - scanner uses the significance of the current character to decide what to do
- Compiler terminology!
- May be useful to parse files
  - Handle C and C++ style comments

Cloning

Object Variable Copying

- When making a copy of an object variable, both the original and the copy refer to the same object.
- If we change the object one of these object variables refers to, the object the other variable refers to is also changed
  - They are the same object

Object Cloning

- To make a new object, clone the object
  - clone starts in the same state as the current object but is a different object

```
Chicken copy = (Chicken)original.clone();
copy.feed();
// original remained unchanged (hasn't eaten)
```

The Protected clone() Method

- clone() method is inherited from the Object superclass
  - protected
  - only Chicken objects, subclasses, and members of package can clone Chicken objects
- Object class does not know the actual structure of its derived classes
  - Derived classes: every class in the Java language

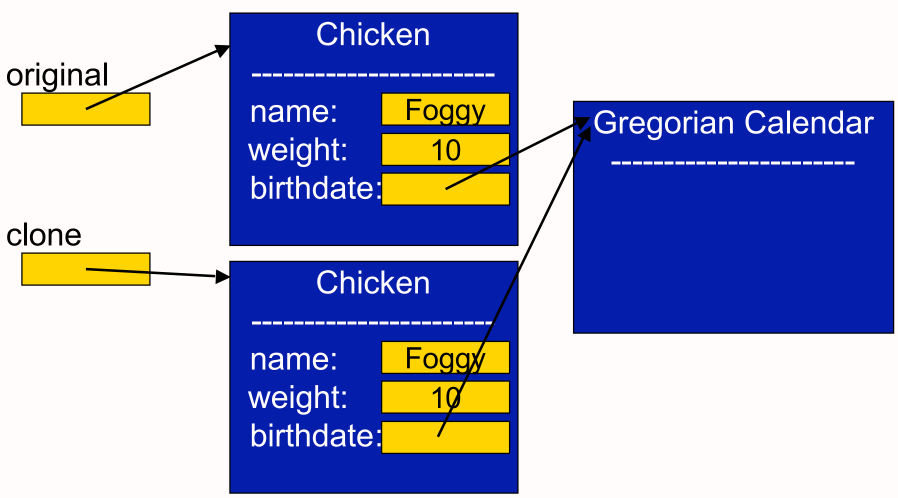
A Problem?

- clone() method makes a field-by-field copy of the object being cloned.
  - OK if the cloned object has only primitive types (no objects)
- What happens if we attempt to clone an object that contains another object?
  - What if we add a field for the Chicken's birthdate?

The Problem with Cloning

- Cloning a Chicken object
  - object variable contained in the Chicken object is copied and both the original and new objects have references to the same object.
- If we change the GregorianCalendar field of the cloned object, we change the original object

The Problem with Cloning



Not a Problem Sometimes…

- We can see the default cloning object is considered shallow
  - Does this matter?
- Some objects are immutable
  - cannot be changed, read-only
  - String and Date objects
  - Shallow copy is okay if the object inside the object to clone is immutable

A Solution to the Problem

- If have mutable objects
  - the clone() method must be overridden
  - make a deep copy
    - Copy subobjects as well.
- Example
  - Copy the GregorianCalendar birthdate object

Object Cloning

- For each programmer-defined class, you should decide if:
  - The default (shallow) clone() behavior is good enough for your class to use
  - The default clone() method can be "made deep" by redefining the clone() method to clone() subobjects as well
  - the class of objects should not be cloned

Object Cloning

- Default: class should not be cloned
- If you choose either of the first two options, you need to do two things:
  - The class must implement the Cloneable interface
    - Marker interface
  - The class must redefine the clone() method with the public access modifier
    - allows objects to be cloned by any class/object
    - you can make an overridden method less private but not more private

Implementing the clone() Method

- If a class is marked as Cloneable, redefine clone()
  - even if you want the default shallow copy

```
class Person implements Cloneable
{
      public Object clone()
      {
            try {
                  return super.clone()
            }
            catch (CloneNotSupportedException e)
            { return null; }
      }
}
```


Implementing the clone() Method

```
class Chicken implements Cloneable
{
      public Object clone()
      {
        try {
        // call Object.clone()
        Chicken cloned = (Chicken)super.clone();

        // clone mutable fields
        cloned.birthdate =
             (GregorianCalendar)birthdate.clone();
        return cloned;
      }
      catch (CloneNotSupportedException e)
        { return null; }
      }
}
```

Collections

June 22, 2006 Sara Sprenkle - CISC370 23

Collections

- Similar to C++ Standard Template Library
- Also known as Containers
- group multiple elements into a single unit
- store, retrieve, manipulate, and communicate aggregate data
- represent data items that form a natural group
  - poker hand (a collection of cards)
  - mail folder (a collection of letters)
  - telephone directory (a mapping of names to phone numbers).
- Examples: Hashtables, Sets, Vector

June 22, 2006 Sara Sprenkle - CISC370 24

Collections Framework

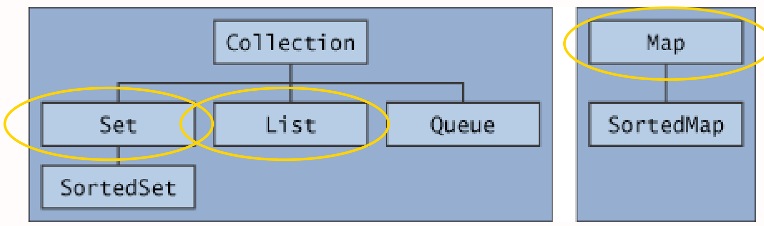
- a unified architecture for representing and manipulating collections
- More than arrays
  - More flexible, functionality, dynamic sizing
- java.util

Collections Framework

- Interfaces
  - abstract data types that represent collections
  - collections can be manipulated independently of implementation
- Implementations
  - concrete implementations of the collection interfaces
  - reusable data structures
- Algorithms
  - methods that perform useful computations on collections, e.g., searching and sorting
  - polymorphic: same method can be used on many different implementations of the appropriate collection interface
  - reusable functionality

Core Collection Interfaces

- Encapsulate different types of collections



Generic Collection Interfaces

- New to 1.5: Generic Collections
  - declaration of the Collection interface:

```
public interface Collection<E>...
```

    - <E> means interface is generic for element class
- specify the type of object when declare a Collection
  - allows the compiler to verify that the type of object you put into the collection is correct
    - reduces errors at runtime
- Example, a hand of cards

```
List<Card> hand = new List<Card>();
```

Make sure put in, get out appropriate type

`List` Interface

- An ordered collection of elements
- Can contain duplicate elements
- Has control over where objects are stored in the list
- `boolean add(Object o)`
  - Boolean so that List can refuse some elements
    - e.g., refuse adding null elements
- `Object get(int index)`
  - Returns elements at the position index
- `int size()`
  - Returns the number of elements in the list
- And more! (contains, remove, toArray, …)

`List` Implementations

- `ArrayList`
  - Resizable array
  - Used most frequently
  - Fast
- `LinkedList`
  - Use if adding elements to beginning of list
  - Use of often delete from middle of list

cards.Deal.java

Implementation vs. Interface

- Implementation choice affects only performance
- Preferred style
  - choose an implementation
  - assign the new collection to a variable of the corresponding interface type
    - or pass the collection to a method expecting an argument of the interface type
- Why?
  - Program does not depend on methods in a given implementation
  - Programmer can change implementations
    - performance concerns or behavioral details

`Set` Interface

- No duplicate elements
  - Needs to be able to determine if two elements are “logically” the same (equals method)
- Models mathematical set abstraction
- `boolean add(Object o)`
  - Boolean so that Set can refuse some elements
    - e.g., refuse adding null elements
- `int size()`
  - Returns the number of elements in the list
- Note: no get method -- get #3 from the set?
- And more! (contains, remove, toArray, …)

`Set` Implementations

- `HashSet`
  - Hash table
  - Used more frequently
  - Faster than TreeSet
  - No ordering
- `TreeSet`
  - Tree
  - Sorts

FindDuplicates.java

`Map` Interface

- Maps keys to values
- No duplicate keys
  - Each key maps to at most one value
- `Object put(Object key, Object value)`
  - Returns old value that key mapped to
- `Object get(Object key)`
  - Returns value at that key
- `Set keySet()`
  - Returns the set of keys

`Map` Implementations

- HashMap
  - Fast
- TreeMap
  - Sorting
  - Key-ordered iteration
- LinkedHashMap
  - Fast
  - Insertion-order iteration
  - Remove stale mappings --> custom caching

Declaring Maps

- Declare types for both keys and values
- `Class HashMap<K,V>`

```
Map<String, List<String>> map
    = new HashMap<String, List<String>>();
```

Keys are Strings
Values are Lists of Strings

Traversing Collections (1)

- For-each loop:

```
for (Object o : collection)
    System.out.println(o);
```

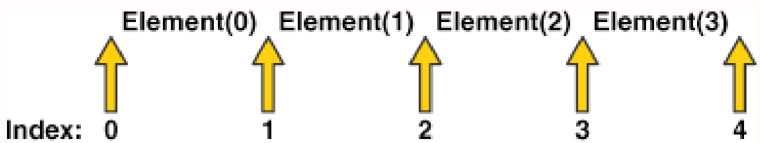
- Valid for all Collections
  - Maps (and its subclasses) are not Collections
  - But, Map's keySet() is a Set and values() is a Collection

Traversing Collections: Iterators

- Java Interface
- Same idea as C++ iterators
- `Object next()`
  - get the next element
- `boolean hasNext()`
  - are there more elements?
- `void remove()`
  - remove the previous element
  - Only *safe* way to remove elements during iteration
    - Not known what will happen if remove elements in for-each loop

Iterator: Like a Cursor

- Always between two elements

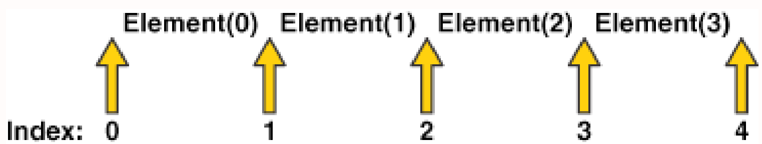


Polymorphic Filter Algorithm

```
static void filter(Collection c) {
  Iterator i = c.iterator();
  while( i.hasNext() ) {
     // if the next element does not
     // adhere to the condition, remove it
     if (!cond(i.next())) {
         i.remove();
     }
  }
}
```

Traversing Lists: List Iterator

- Methods to traverse list backwards
  - `listIterator(int position)`
  - Pass in size() as index to get at end of list
  - `hasPrevious()`
  - `previous()`
- Used for insertion/modification/deletion in linked lists in the middle



Enumeration

- Legacy class
- Similar to Iterator
- `boolean hasMoreElements()`
- `Object nextElement()`
- Longer method names
- Doesn't have remove operation

Collection classes to avoid

- Synchronized classes
  - For multiple threads sharing same collection
  - Slow down typical programs
  - e.g., Vector, Hashtable
  - See java.util.concurrent

Utility Class: Collections

- Similar to `Arrays` class
- Contains methods for
  - Binary searching
  - Sorting
  - Min/max finding ("extremes")
  - Reversing
  - Shuffling
  - …

Alternative Sorting

- What if object is Comparable but does not sort the way you want?
  - Special case
    - Don't want to change class
    - Don't have access to class
  - e.g., sort strings so capital, lowercase letters are the same
- Use Comparator interface

`Comparator` Interface

- Declares two methods:
  - `int compare(Object o1, Object o2)`
    - compare two objects and return a value as if we called `o1.compareTo(o2)`
  - `boolean equals(Object other)`
    - check to see if this Comparator equals other
- Overloaded versions of `sort` in Arrays and Collections
  - Arrays: `void sort(Object[] array, Comparator c)`
  - Collections: `void sort(Collection col, Comparator c)`

ChickenComparator.java

Localization/Internationalization

- Part of java.util
- Customize how data is presented and formatted
- Use Locale objects
  - Specify language, geographic region
- Calendar, GregorianCalendar
- Currency
- Date
- TimeZone

Compression

Compression

- Reduce the size of files
  - While **not** losing data!
  - Easier to transport over the network
- Often used in conjunction with archival
  - Archive: merge multiple files into one file
- In our assignment instructions in UNIX
  - Use tar to archive the assignment (assignx.tar)
  - Use gzip to compress the assignment (assignx.tar.gz)

Compression: java.util.zip

- GZIP compression
  - GZIPInputStream
  - GZIPOutputStream
  - Standard filtered stream
    - you don't do anything special!

ZIP files

- ZIP files
  - Both archival and compression
  - Used in WinZip
  - Supports encryption
- Tar/GZIP typically gets better compression
  - i.e., smaller files
  - Better to zip all together rather than zip one file at a time
- ZIP allows random access to file

ZIP files

- Each file within a ZIP archive is represented using a ZipEntry
- Set the filename of a ZipEntry using a contructor
- Get the name and uncompressed size using the `getName()` and `getSize()` methods

Reading Zip Files

- Method 1: `ZipFile` class
  - Create a ZipFile object for your file
    - pass it the File or a String
  - Get an Enumeration containing instances of `ZipEntry` with **entries()**
  - Get an InputStream for a single entry by calling **getInputStream(ZipEntry ze)**

Reading Zip Files

- Method 2: **ZipInputStream** class
  - Create a ZipInputStream
  - Connect it to an existing file stream
  - Read the entries in sequence:
    - Get a reference to the next ZipEntry by calling **getNextEntry()**
    - Use the ZipInputStream to read from this entry
      - it returns -1 at the end of the entry rather than the zip file
      - close the entry with **closeEntry()**

Writing ZIP files

- Use the ZipOutputStream class
- Like the inverse of ZipInputStream:
  - putNextEntry()
  - Typical OutputStream methods
  - closeEntry()

3 Week Checklist

- Primitive types
- Object-oriented concepts
- Lots of I/O
  - Parsing
- Lots of Collections
- Serialization
- Compression
- Helper methods: sorting, searching made easy
- Your job: representing data, leverage classes

Assignment 3

- Applying streams and collections to your media library
- Code submission
  - New versions of your classes --> New package