Collections & Implementaitons

Interfaces, Classes, Iterators, JavaDoc, and Testing

CSE 373

Data Structures

Winter 2007

Agenda

- Review of containers (ADTs) and implementations
- Running example – list collection with two implementations: arrays and linked list
- Java best practices
  - Interfaces and classes
  - Iterators
  - JavaDoc
  - Junit testing

Types and Implementations

- Common collection types
  - List, queue, stack, set, bag (multiset), priority queue, map/dictionary, graph
- Variations: sorted or not (sets, maps, others)
- Implementation techniques
  - Array, linked list (many variations), hashing, trees/ graphs (many, many variations), heaps
- Is it a collection or an implementation technique? Might be either depending on context, e.g., trees, graphs

Running Example: Lists

- An ordered collection, position matters
- Operations
  - Constructor: create a properly initialized empty list
  - Modifications: clear, add/remove element at end or at position, change element
  - Queries: size, isEmpty, find/get element
  - Processing: iterator

Java

- CSE373 is about data structures, not Java, but…
- Java and the culture around it capture many "best practices", so…
- We'll learn those practices and use them when appropriate

Abstractions in Java

- Every interface and class defines a type
- Conventions
  - Define *every* important type with an interface
  - Provide implementations as appropriate
  - Client code should use the interface type name instead of a specific implementation unless there is a good reason not to
    - Promotes generality and reusability

Today's Example

- Interfaces: BasicList, BasicListIterator
  - › Specifies list operations essentially the same as ones in Java collection classes
    - (Except not using generics)
- Implementations: BasicArrayList, BasicLinkedList
  - › Particular implementations using array and linked lists as the backing store
- Sample code on the web

BasicArrayList Representation

- Representation is an array and count of number of items currently stored

  private Object[ ] items;

  private int nItems;
- Invariant
  - References to objects in the collection are stored in items[0..nItems-1]
  - › Check invariants while coding – powerful bug avoidance tool

Comments

- Java comments

  // to end of line

  /* c-style */

  /** JavaDoc */
- All comments should capture "why" that is not apparent from the "how" of the code
- JavaDoc – a particular style of comments that can be automatically processed to create documentation
  - › First used to document the standard Java libraries

JavaDoc

- Can put almost any html between /** and */
- Place right before interface/class or method definitions (and elsewhere if wanted, but these are the main uses)
- Special tags to identify particular things

  @author, @version – primarily for classes/ interfaces

  @param, @return, @throws – primarily methods

Using JavaDoc

- Every class/interface should have a summary JavaDoc comment at the beginning
- Every public method (visible outside the class) should use JavaDoc to explain *all* parameters, return values, exceptions that are part of the method contract
- Exception: JavaDoc automatically copies comments from interfaces to doc pages for implementing classes – no need to duplicate

Exceptions

- Problem: a collection (or other object) may be in a position to detect an error but not know how best to handle it
- Solution: throw an exception object that can be caught to handle the error or, if not caught, will terminate the program

  throw new IndexOutOfBoundsException();

Exception Guidelines

- Extensive hierarchy of exception types in Java standard library – use one of these if appropriate; define your own if library ones don't meet your needs
- Throw the most specific exception appropriate to the error, e.g., IllegalArgumentException(…) instead of Exception(…)
- Optional argument: string that provides detail

```
throw new IllegalArgumentException("null not allowed…");
```


Processing Collection Contents

- To process an *ordered* collection we can access the elements by position

```
for (int k = 0; k < size; k++)
    do something with things.get(k)
```

- But
  - › This may be inefficient if access by position is not guaranteed to be fast
  - › Likely impossible (e.g., no get(k)) for unordered collections (sets, maps)

Iterators – General Solution

- Every Java collection can provide iterators that can be used to access its contents

```
Iterator it = things.iterator();
while (it.hasNext()) {
    Object item = it.next();
    process item
}
```


Standard Iterator Methods

- Forward access

  hasNext() – true if more elements

  next() – return next element and advance
- Similar methods for reverse access in some collections (e.g., lists)
- Modification

  remove() – remove last item returned by next/previous

Iterator Details

- Multiple iterators may be active on a single collection at the same time
- Remove may only be used once per next/ previous, otherwise IllegalStateException thrown
- Collection may not be modified while iteration is in progress except by remove; ConcurrentModificationException thrown if next/remove/previous attempted after other modification, including remove() in other iterator(s)

Iterator Implementation

- Iterators typically need access to internal, private implementation details of associated collection class
- Clean solution: nest the iterator class inside the container class
  - › Should be private – only outside access is via the collection's iterator() method that returns a new instance

Second List Implementation

- Same interfaces: BasicList, BasicListIterator
- Implementation: BasicLinkedList
  - › Implemented with a single-linked list as the backing store
  - › Also appears “infinite” to clients

BasicLinkedList Nodes

- Each link in the list is an instance of the following nested (local) class

```
private class Link {
  public Object item;   // list element referenced
                        //   by this link
  public Link   next;   // next link or null if this is the last
                        //   link in the list

  // constructor for convenience
  public Link(Object item, Link next) { … }
}
```


List Representation

- We can implement a BasicLinkedList with (only) the following instance variable

```
private Link head;    // reference to first link in
                      // the list, or null if the
                      // list is empty
```

  - › (Of course, additional instance data may make it easier to do some things faster, but this is enough to get started.)

Typical List Operation

```
public int indexOf(Object obj) {
  // sequential search
  int pos = 0;      // position of current link in the list
  Link p = head;
  while (p != null) {
    if (p.item.equals(obj)) {
      return pos;
    }
    p = p.next;
    pos++;
  }
  return -1;
}
```


Another List Operation

```
public int size() {
  // count the number of links in the list
  int nItems = 0;
  Link p = head;
  while (p != null) {
    nItems++;
    p = p.next;
  }
  return nItems;
}
```

- But wait!! This takes O(n) time!!! We should be able to do better – and we can

Speeding up size()

- Instead of counting the links, keep the list length in a separate instance variable, updated as needed
- A typical example of trading storage for computation
- But how do we verify that we don’t break anything if we make this change?
  - › And how do we know that things are ok to start with?

Testing & Debugging

- Testing
  - › Verify that things work as expected
  - › Be able to reverify as software evolves
- Debugging
  - › Controlled experiment to discover what is wrong and where

Testing Strategies

- Test "typical" cases – basic functional tests
  - › Do operations work properly on a non-empty list?
- Test "edge" cases
  - › Zero, one, many (empty list, single element, more, …)
  - › Limit cases – what happens if a container is full
  - › Error cases – do things blow up as expected (index out of bounds, other exceptions)
- Stress tests – harder, but needed in production code – e.g., large workloads

Debugging Strategies

- Questions to ask
  - › What's wrong?
  - › What's working? How far do we get before something fails?
  - › What are the symptoms?
  - › What changed since the last time it worked?
- Observing strategies
  - › Print statements(!)
  - › Debuggers – CAT scans for software
  - › Etc…

Unit Tests

- Idea: a collection of tests for individual operations
- Effective testing: lots of small tests, each of which checks something specific
  - › Incremental building and testing
  - › Avoid "big-bang" tests as your only strategy

Where to Put Tests

- Type them in using the programming environment (tedious)
- Lots of test programs (better – don't have to retype – but still tedious to run repeatedly)
- Automated test frameworks
  - › Been around for a while, but popularized by "extreme programming" / "agile development" movements in recent years

JUnit

- Test framework for Java unit tests
- Implemented as classes that extend Junit's TestCase class
  - › Need to import junit.framework.*
- Key: test methods are named testXXXX
- Optional: setUp() method to create state before each individual test is run
- More, but these are the core ideas

Inside Test Methods

- Inherited from TestCase; typical ones include

```
assertEquals(expected, actual)
assertEquals(expected, actual, delta) // doubles
assertTrue(condition)
assertFalse(condition)
assertNull(object)
assertNotNull(object)
Fail("message")  // generate failure if control
                 //  should not reach a particular point
                 //  (example: expected exception not thrown)
```


Unit Test Strategy

- Define tests before or as you write code
- Add and run tests each time you add something small to the code
- Rerun tests to verify nothing broken after changes
- If a bug is detected, create a test to demonstrate it, fix it, then keep the test forever as part of the test suite