Exceptions, Collections, and Lists
Recursion, Stacks, Queues

October 2-3 2002

Neil Ernst

Reading Assignment
Chapters 4-5

Outline

- ***Questions from last time***
- ***Reading assignment – chap. 4 and 5 in the text***
- ***Today:***
  - ✍ ***Exceptions***
  - ✍ ***Collections in Java***
  - ✍ ***Dynamic data structure: Linked List***
- ***Tomorrow:***
  - ✍ ***Dynamic data structure: Stack***
  - ✍ ***Dynamic data structure: Queue***
  - ✍ ***Iterators (depending on time)***

Interface example: linear search

- Practical example of why we use interfaces
- Knowing something implements an interface allows us to make certain assumptions about that class
- Code walkthrough

Exceptions

Why are Exceptions in Java?

- ***Basic philosophy of Java is to minimize chances for programmers to make mistakes***
  - ✍ ***At expense of flexibility (C/C++)***
  - ✍ ***e.g. type-checking, garbage collection***
- ***Best time to prevent errors is at compile time***
- ***But of course this isn't always the case….. e.g. null references are hard for compiler to identify***
- ***So what do we do when we encounter things we don't expect?***
- ***Let the program crash?***

Exceptions

- Exceptions provide a convenient way to handle abnormal conditions
- Could be used to handle problems and continue on – *resumption vs. termination*
  - ✍ But, this can be difficult to anticipate
  - ✍ New error conditions need to be handled
- Terminology:
  - ✍ *throw* an exception to indicate a problem
  - ✍ *catch* an exception to deal with the problem
  - ✍ *finally* do something, whether an exception happened or not
- Naturally, exceptions are objects.

Exceptions

- ✍Thrown exceptions bypass the normal method call-return mechanism
  - a method that throws an exception does not return a value (but it will return a reference to an exception object!)
  - a thrown exception may propagate out through multiple layers of called methods

- ✍Exceptions can be….
  - thrown by either the Java Virtual Machine or the program
  - caught by the program — if the VM catches one, it's a crash!

What happens when an exception is thrown?

- ***First an exception object is created***
- ***The current path of execution is halted***
- ***Next the Java exception handling mechanism tries to find an appropriate place to continue executing the program***
  - ✍ ***That is, an exception handler for that type of exception***
- ***If no exception handler (catcher) is immediately found the reference to the exception object is ejected up a calling level in your program and so on until a handler is found***
- ***If there is no obvious handler in your code, the JVM will halt execution (crash!).***

How to Throw Exceptions

- ***Use the throw statement, with an exception object as argument***
  - ✍ e.g. `if` <*some error*> `throw new Exception();`
- ***You almost always want to create a new instance of the exception (sometimes you may wish to reuse an exception object and rethrow it)***
- ***Unless the exception is caught in the same method or is unchecked, your method must declare that it might throw this exception***
  - ✍ e.g.

```
public myMethod( arg1...) throws
  Exception { ... }
```

How to Catch Exceptions

- ***Use the try-catch-finally statement***
- ***`try { }`***
  - ***put code that may throw exceptions here***
  - ***also any code that needs results from code above***
- ***`catch (AnExceptionClass e) { }`***
  - ***deal with errors of kind AnExceptionClass here***
  - ***the parameter e will contain the exception object***
- ***`finally { }`***
  - ***put code here that you want to execute after the try block whether an exception was thrown or not (and whether it was caught or not)***
  - ***cleanup code***

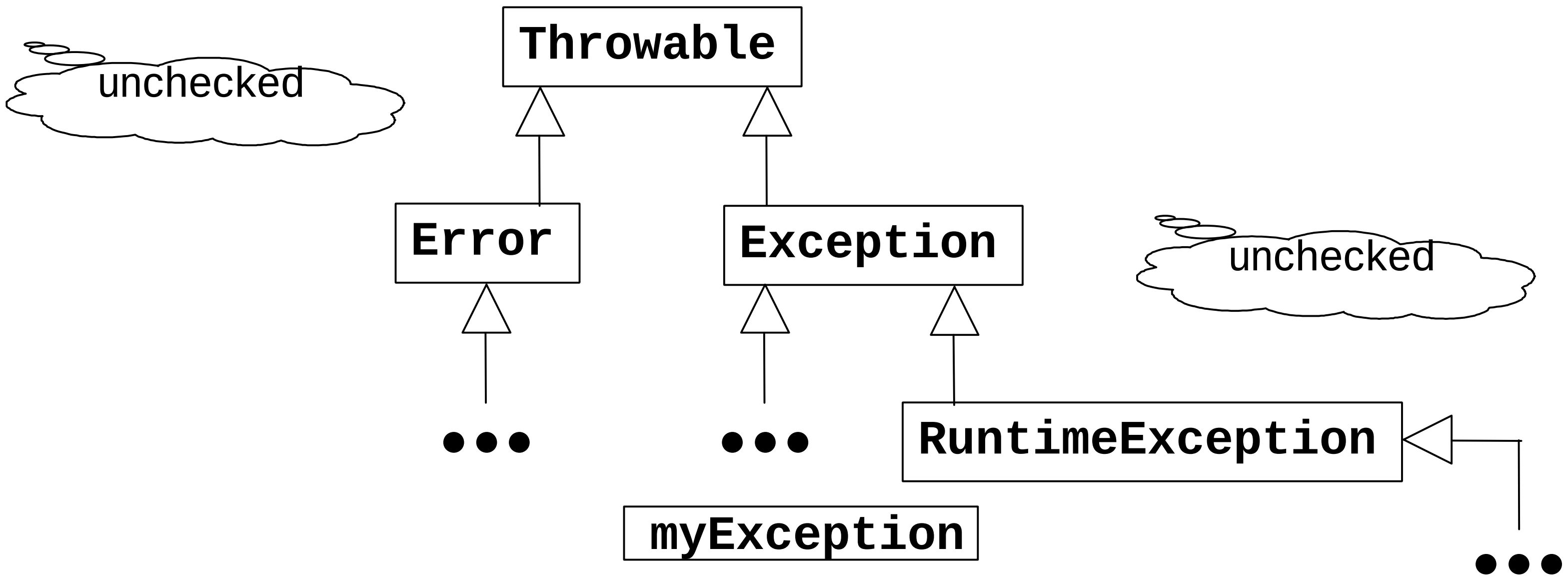
Example of a Catch

- Here we'll be catching an exception generated by the virtual machine:

<Eclipse code>

Exceptions Hierarchy

- ***An exception is just an object, but:***
  - ***all exception classes must derive from Throwable***
  - ***problems at the virtual machine level are Errors, and should almost never be caught***
  - ***all user (and many system) exception classes derive from Exception***
  - ***unchecked exception classes derive from RuntimeException***

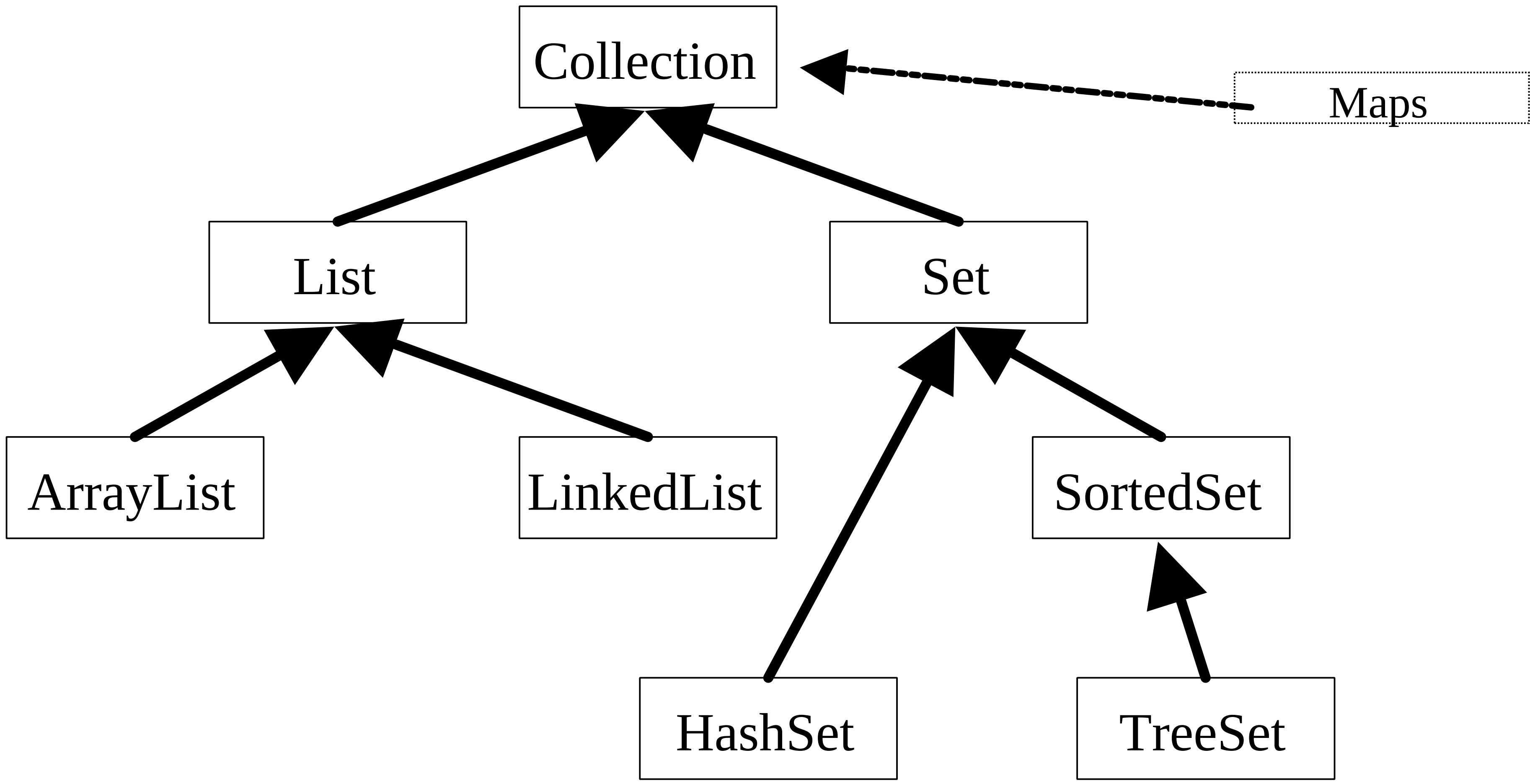


Collections

Why a Collections Framework

- Almost all programs need storage space
- Few know exactly how much to allocate until run-time
- Arrays are good (speedy, small) at:
  - ✍ holding relatively fixed amounts of identical type data, e.g. a list of students in a class – changes are small
- Arrays are bad at:
  - ✍ inserting and repositioning data
  - ✍ expanding and contracting as needed
  - ✍ holding data of different types
- Collections other than arrays provide this flexibility
- Sometimes different tradeoffs are useful
  - ✍ speed, access time, permitted operations
- Therefore, number of different collection classes
- In practice, often end up using one type more than others

Java Collections Hierarchy



Types of Collections

- ***Almost all collections are defined to contain objects***
  - ✍ ***As a result, any object can be put into a collection***
  - ✍ ***Use the `instanceof` operator to determine the kind of object during retrieval***

    ```
    x instanceof String
    ```

  - ✍ ***Cast the instance to the correct type accordingly***

    ```
    String s = (String)x;
    ```

- ***Collections are heterogeneous or homogeneous***
  - ✍ ***Homogeneous: all components are of the same type***
  - ✍ ***Heterogeneous: components may be of different types***
- ***Most collections in Java are heterogeneous***

Wrappers for primitive types

- **Primitive types (i.e., `byte, short, int, long, float, double, char, boolean`) are not compatible with the reference type `Object`**
- **Thus, values of primitive cannot be passed to parameters of type `Object`**
- **To get around this problem, Java provides wrapper classes for all primitive types**

```
public final class Integer implements Comparable {
  private int value;
  public Integer(int x) { value = x; }
  public int intValue() { return value; }
  public String toString() { return "" + value; }
  public int compareTo() { … }
}
Integer k1 = new Integer(17);
```

Collections

- Collections, data structures, abstract data types (ADTs) consist of two parts
  - data representation
  - operations on those data
- Java provides an entire set of collection APIs
  - interfaces and implementations for fundamental data structures such as lists, stack, queues, deques, trees, graphs
- A container or dictionary is a special collection which supports the operations member , insert, delete, isEmpty
- Here is a simple container interface

```
public interface Container {
  Object member(Object x);
  void insert(Object x);
  Object delete(Object x);
  boolean isEmpty();
}
```

The Java Collection interface

```
public interface Collection {
  boolean add(Object x);
  boolean addAll(Collection c);
  void clear();
  boolean contains(Object x);
  boolean containsAll(Collection c);
  boolean equals(Object x);
  int hashCode();
  boolean isEmpty();
  Iterator iterator();
  boolean remove(Object x);
  boolean removeAll(Collection c);
  boolean retainAll(Collection c);
  int size();
  Object[] toArray();
  Object[] toArray(Object[] a);
}
```

Linked Lists

- A list is a collection of data much like an array
- Advantage: easy to resize a list
  - add: create a new Node and update references
  - delete: change the references, Node is garbage collected
  - insert: change references
- Disadvantage:
  - greater space demands (a new object for each node)
  - more complex operations
- What might we want to do with a list?
  - Taken from Java API: insertFirst, insertLast, deleteFirst, deleteLast, isEmpty, add
- Let’s examine the list code
  - <Eclipse>

Outline

- ***Questions from last time***
- ***Today:***
  - ✍ ***List exercise, notion of double-linked list***
  - ✍ ***Recursion***
  - ✍ ***Dynamic data structure: Stack***
  - ✍ ***Dynamic data structure: Queue***
  - ✍ ***Iterators (depending on time)***

Class exercise

- Yesterday we discussed a singly linked-list structure.
- Question: how to insert an element at a certain position in the list?
  - write a pseudocode method `add(Object o, int index)` which takes the object to insert into the data field and an index which is the number of the element after insertion (indexed from zero, like an array).
  - be careful with the references.
  - assume the list is not empty and that we're inserting in the middle (otherwise you need some error conditions).
  - good midterm/final question!

Node and LinkedList

```
public class LinkedList {
private Node head;
private int size;

public LinkedList() {...}
public Node getHead() {...}
public boolean isEmpty()
   {..}
public int size() {..}
public Object getFirst()
   {...}
public void insertFirst
   (Object data) {..}
public String toString()
   {..}
public Object deleteFirst()
   {...}
```

```
public class Node {
  private Object data;
  private Node next;
  public Node(Object data,
       Node next)    {..}
  public Node(Object data) {..}
  public Object getData() {...}
  public Node getNext() {...}
  public void setData(Object data)
       {.. }
  public void setNext(Node
next)        {...}
}
```

Double linked list

- The same as a single-linked list but...
  - ✍ new Node type, with pointers in two directions, `next` and `prev`
  - ✍ reference to end of the list - `tail`
  - ✍ additional methods `insertLast()` and `deleteLast()`

Recursion

text, pp 148-149 (brief)

Recursive algorithms and data structures

- ***A method (algorithm) or class that is partially defined in terms of itself is called recursive***
- ***Recursion is a powerful algorithm design and programming tool that can lead to elegant and efficient algorithms and data structures***
- ***A recursive algorithm consists of***
  - ✍ ***a base case***
  - ✍ ***a recursive call***

Recursive methods and classes

- A recursive method is a method that directly or indirectly calls itself
- Simplest direct and indirect recursive methods; note that both examples result in infinite recursion since there is no base case

```
void a() { a(); } // direct recursion
void a() { b(); } void b() { a(); } // indirect
```

- Shortest and simplest direct and indirect recursive classes

```
class X { X x; } // direct recursion
class X { Y y; } class Y { X x; } // indirect
```

Compute the sum of k integers recursively and iteratively

```
recursiveAlgo(int n)
  if ("simplest case")
    // base case
    "solve directly"
    "for example for n=1"
 else
    // induction step
    "make a recursive call
     with simpler case"
    "for example for n-1"
```

Recursive algorithm

```
int sum(int k) {
  if (k==1) return 1;
  else return sum(k-1) + k;
}
```

- **<- Pseudocode for a recursive method**
- **Base case is a simple case where we know the solution**
- **For the induction step, we assume that we know the solution for a previous solution, say n-1, and compute the solution in terms of this solution**
- **For example, if we know the sum of the first n-1 integers (i.e., sum(n-1)), the sum of n integers is n + sum(n-1)**
- **Iterative algorithm:**

```
int sum(int k) {
  int s = 0;
  for (int j=1; j<=k;
  j++)
    s = s + j;
  return s;
}
```

Stacks and Queues

chap 4.1 and 4.2

Stack interface

```
public interface Stack {
  void push(Object data);
  Object pop();
  Object top();
  boolean isEmpty();
  int size();
}
```

- LIFO (Last-in, first-out) list
- Examples:
  - ✍ Stack of plates in cafeteria
  - ✍ Run-time stack in operating systems
  - ✍ Recursion
  - ✍ Evaluating expressions
  - ✍ Balanced parentheses

Queue interface

```
public interface Queue {
  void enqueue(Object data);
  Object dequeue();
  Object front();
  boolean isEmpty();
  int size();
}
```

- ***FIFO (First-in, First-out) list***
- ***Examples:***
  - ✍ ***Check-out line at store***
  - ✍ ***Car wash***
  - ✍ ***Network queues***
  - ✍ ***Traffic simulation***

Stack and queue definitions

- Interface code defined in Eclipse

Run-time stack

- ***Every recursive algorithm can be converted into a non-recursive or iterative algorithm by simulating the run-time stack***
- ***The run-time stack consists of activation records or stack frames***
- ***An activation record contains the following information***
  - ✍ ***Return address (address of caller)***
  - ✍ ***Destination address (address of callee)***
  - ✍ ***Actual parameters (parameters being passed)***
  - ✍ ***Local variables (local variables of the routine being called)***
- ***Whenever a method call is made, a new activation record is allocated and pushed onto the run-time stack***
- ***When a call returns, its record is popped off the run-time stack***