Java Container Classes
A motivating example

- Android and iOS use an event driven model for programming apps
- Apps *register* for a service by sending an object of type T to the provider of a service (this is one of three ways I know of for apps to get services)
  - T is usually an interface or abstract class
- Service responds at some later time by calling some method m( . . .) on the sent object and passing information through that method
  - Because the registered object is of type T the service knows it implements an m with the right signature
A GPS service - app side

- Phones have GPS devices
- Some Apps would like to be notified by the GPS if the phone moves more than X feet
  - Can set off an alarm
  - Useful to see if people are stealing your phone
  - Useful if you are on an anchored boat
- Registration at a high level looks like code to the left

```java
class appGPSHandler implements GPSUser {
    ...
    void moveAlert(...) {...}
}

class myApp {
    GPSUser handler = ...
    new appGPSHandler();
    ...
    GPS.registerAlarm(handler);
    ...
}
```
A GPS service -- Android side

class GPS {
    static List<GPSUser> l = new ...
    static void register(GPSUser u) {
        l.insert(u);
    }

    static void moved() {
        for each element e in l {
            if movement is enough {
                e.moveAlert(...);
            }
        }
    }
}

• When a GPS request for service is registered it is put into a list l that can hold GPSUser objects
• When the service is to be provided, the element e is pulled out of the list l and the app method moveAlert(...) is called on the element
class appGPSHandler {
    
    void moveAlert(...) {
    }
}

class GPS {
    static void register(...) {
        
        static void moved(...) {
    }
}

class myApp {
    
    GPS.register(handler);
    
    
}
class appGPSHandler {
    ... 
    void moveAlert(...) {
        ...
    }
    ...
}

class myApp {
    ...
    GPS.register(handler);
    ...
}

class GPS {
    ...
    static void register(...)
    static void moved(...)
    ...
}

List l
handler
class appGPSHandler {
    void moveAlert(...) {
    }
}

class myApp {
    GPS.register(handler);
}

class GPS {
    static void register(...) {
    }
    static void moved(...) {
    }
}
```java
class appGPSHandler {
    . . .
    void moveAlert(. . .) { . . . };
    . . .
}

class myApp {
    . . .
    GPS.register(handler);
    . . .
}

class GPS {
    . . .
    static void register . . .
    . . .
    static void moved . . .
    . . .
}

List l
handler.moveAlert(
```
```java
class appGPSHandler {
    . . . 
    void moveAlert(. . .) { . . .};
    . . .
}

class myApp {
    . . .
    GPS.register(handler);
    . . .
}

class GPS {
    . . .
    static void register . . .
    . . .
    static void moved . . .
    . . .
}

List l

handler.moveAlert();
```
class appGPSHandler
{
    . . .
    void moveAlert(. . .) {. . .};
    . . .
}

class myApp
{
    . . .
    GPS.register(handler);
    . . .
}

class GPS
{
    . . .
    static void void register . . .
    static void moved . . .
    . . .
}

List $l$

handler.moveAlert( )
Abstract classes/interfaces and containers work together

• Interfaces/abstract classes force a derived class to implement functionality
  – This in turn lets users of the object know that the functionality exists in the object
• Containers supply efficient support for holding and accessing lots of objects
• Polymorphism allows
  – objects of many types to be held in a container with some base type (either an interface, base abstract class or non-abstract base class)
  – Implementations of methods in the final class to be called
In our example

• The moveAlert class can do different things based on the app.
• If a mapping app, move a marker on a map every 100 feet
• If a boat anchoring app, sound an alert
• If a tracking app, add an entry to the track log
• All of these can be done from objects in the container as long as the actual derived class contains the proper implementation
• The call to moveAlert( ) doesn't care which, only that the signature is correct
Why Container Classes?

• Many programs use arrays, vectors, lists, queues, stacks, sets to store information.

• Both C++ and Java provide container classes that automatically manage memory, i.e. they allocate additional memory when more elements are added.

• The supported container classes greatly reduce the amount of code and programming needed and improve productivity.

• Container classes and OOP are closely related:
  – Containers hold objects of different derived classes
  – Polymorphism properly invokes the correct methods
Container Class (For Code Reuse)

• A container needs to be able to hold items of different types (i.e. classes). Examples
  – list of strings, integers, floating points, student objects
  – queues of undergraduates, graduate students, staff and faculty
  – maps: name $\rightarrow$ address, student ID $\rightarrow$ name, course title $\rightarrow$ classroom

• C++ standard template library (STL) and Java container classes provide such functionality.
Selecting a container class

• random or sequential accesses?
• allow unique or duplicate items?
• O(1) or O(N) for array-like access (using [index])
• efficient insert / delete?
  – front
  – end
  – middle
• Java containers cannot store primitive types (int, char, float ...), they can store objects only. Primitive types, however, have corresponding object types (e.g. Integer, Boolean) that can be held in containers.
• C++ containers can store primitives.
# Efficiency

<table>
<thead>
<tr>
<th>operation</th>
<th>vector</th>
<th>deque</th>
<th>list</th>
</tr>
</thead>
<tbody>
<tr>
<td>array-like access</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>insert/delete at front</td>
<td>$O(N)$</td>
<td>$O(1)+$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert/delete at end</td>
<td>$O(1)+$</td>
<td>$O(1)+$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert/delete in middle</td>
<td>$O(N)$</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

N: current number of items
Two suggestions when using containers

• If code you are writing can ever exist in a multithreaded environment
  – Make sure the container is thread safe or add your own synchronization
  – Make sure actions on objects stored in the container are thread safe
• If you have the choice of using a Java or C++ container or writing your own, use the supplied one
  – Even if yours and their’s are both O(N), there constant will almost certainly be smaller than yours
  – If thread safe, smart people will have spent lots of time tuning this to avoid unnecessary synchronization
Java Containers
Java List
java.util

Interface List<E>

All Superinterfaces:

Collection<E>, Iterable<E>

All Known Implementing Classes:

AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector

public interface List<E>
extends Collection<E>

An ordered collection (also known as a sequence). The user of this interface has precise control over where in the list each element is inserted. The user can access elements by their integer index (position in the list), and search for elements in the list.

Unlike sets, lists typically allow duplicate elements. More formally, lists typically allow pairs of elements e1 and e2 such that e1.equals(e2), and they typically allow multiple null elements if they allow null elements at all. It is not inconceivable that...
Interface and Class

• A Java interface serves as an *abstract class* and cannot be instantiated.
• An interface can be implemented by classes.
• Typically, an interface is a common base for several related classes, for example, interface *List* as the base of *ArrayList, LinkedList, Stack, and Vector.*
### Method Summary

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>boolean add(E e)</code></td>
<td>Apps the specified element to the end of this list (optional operation).</td>
</tr>
<tr>
<td><code>void add(int index, E element)</code></td>
<td>Inserts the specified element at the specified position in this list (optional operation).</td>
</tr>
<tr>
<td><code>boolean addAll(Collection&lt;? extends E&gt; c)</code></td>
<td>Apps all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection’s iterator (optional operation).</td>
</tr>
<tr>
<td><code>boolean addAll(int index, Collection&lt;? extends E&gt; c)</code></td>
<td>Inserts all of the elements in the specified collection into this list at the specified position (optional operation).</td>
</tr>
<tr>
<td><code>void clear()</code></td>
<td>Removes all of the elements from this list (optional operation).</td>
</tr>
<tr>
<td><code>boolean contains(Object o)</code></td>
<td>Returns true if this list contains the specified element.</td>
</tr>
<tr>
<td><code>boolean containsAll(Collection&lt;?&gt; c)</code></td>
<td>Returns true if this list contains all of the elements of the specified collection.</td>
</tr>
<tr>
<td><code>boolean equals(Object o)</code></td>
<td>Compares the specified object with this list for equality.</td>
</tr>
</tbody>
</table>
```java
public class Main {

    public static void main(String[] args) {

        List<String> animals = new ArrayList<String>();
        // TOP code application logic here
        animals.add("cheetah");
        animals.add("lion");
        animals.add("cat");
        animals.add("fox");
        animals.add("cat"); //duplicate cat
        System.out.println(animals); //cheetah, lion, cat, fox, cat

        animals.remove("lion");
        System.out.println(animals); //cheetah, cat, fox, cat

        animals.add(0, "lion");
        System.out.println(animals); //lion, cheetah, cat, fox, cat

        animals.add(3, "raccoon");
        System.out.println(animals); //lion, cheetah, cat, raccoon, fox, cat
    }
}
```
What is an array list?

• Is it an an Array or is it a list?
• From http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html
  – Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null.
  – In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.
  – (This class is roughly equivalent to Vector, except that it is unsynchronized.)
```java
animals.remove(3);
System.out.println(animals); //lion, cheetah, cat,
//fox, cat

Collections.sort(animals);
System.out.println(animals); //cat, cat, cheetah,
//fox, lion

List pets = new LinkedList();
pets.add("cat"); // (I)
pets.add("dog");
pets.add("bird");
System.out.println(pets); //cat, dog, bird

animals.addAll(3, pets);
System.out.println(animals); //cat, cat, cheetah,
//cat, dog, bird, fox,
//lion

ListIterator iter = animals.listIterator();
while (iter.hasNext()) {
    System.out.println(iter.next()); // (M)
}
```
Iterators

• Iterators are easy ways to traverse a collection of objects
• To be safe, unless allowed or specified by the documentation:
  – Don’t assume an order for how objects are visited
  – Don't change what is being iterated on - be especially careful of adds and deletes
  – Don't assume iterators are thread safe
    • CopyOnWriteArrayList is
    • Vector iterator is not
Java Set
```java
public static void main(String[] args) {
    // TODO code application logic here
    Set<String> animals = new TreeSet<String>();

    animals.add("cheetah");
    animals.add("lion");
    animals.add("cat");
    animals.add("elephant");
    animals.add("cat"); // duplicate cat
    System.out.println(animals);
    // cat cheetah elephant lion
    System.out.println(animals.size()); // 4

    animals.remove("lion");
    System.out.println(animals); // cat cheetah elephant

    Iterator iter = animals.iterator();
    while (iter.hasNext()) { // (L)
        System.out.println(iter.next());
    } // (M)
} // cat cheetah elephant
```
public class TreeSet  
extends AbstractSet  
implements SortedSet, Cloneable, Serializable

This class implements the Set interface, backed by a TreeMap instance. This class guarantees that the sorted set will be in ascending element order, sorted according to the natural order of the elements (see Comparable), or by the comparator provided at set creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the basic operations (add, remove and contains).

Note that the ordering maintained by a set (whether or not an explicit comparator is provided) must be consistent with equals if it is to correctly implement the Set interface. (See Comparable or Comparator for a precise definition of consistent with equals.) This is so because the Set interface is defined in terms of the equals operation, but a TreeSet instance performs all key comparisons using its compareTo (or compare) method, so two keys that are deemed equal by this method are, from the standpoint of the set, equal. The behavior of a set is well-defined even if its ordering is inconsistent with equals; it just fails to obey the general contract of the Set interface.

Note that this implementation is not synchronized. If multiple threads access a set concurrently, and at least one of the threads modifies the set, it must be synchronized externally. This is typically accomplished by synchronizing on some object that naturally encapsulates the set. If no such object exists, the set should be "wrapped" using the Collections.synchronizedSet method. This is best done at creation time, to prevent accidental unsynchronized access to the set:

    SortedSet s = Collections.synchronizedSortedSet(new TreeSet(...));

The iterators returned by this class's iterator method are fail-fast: if the set is modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException. Thus, in the face of concurrent modification, the iterator fails quickly and cleanly, rather than risking arbitrary, non-deterministic behavior at an undetermined time in the future.

Note that the fail-fast behavior of an iterator cannot be guaranteed as it is, generally speaking, impossible to make any hard guarantees in the presence of unsynchronized concurrent modification. Fail-fast iterators throw ConcurrentModificationException on a best-effort basis. Therefore, it would be wrong to write a program that depended on this exception for its correctness: the fail-fast behavior of iterators should be used only to detect bugs.

This class is a member of the Java Collections Framework.

Since:
1.2
See Also:
Collection, Set, HashSet, Comparable, Comparator, Collections.synchronizedSortedSet(SortedSet), TreeMap, Serialized Form
no duplicate element in Set elements sorted
Java Map
Map (Hash Table)

- array: integer → element (object)
- map: key (object, integer, or string ...) → value (object)
- example:
  - name → phone number
  - student ID → department
  - city name → zip code
- **Keys must be unique** and do not have to be contiguous (as is required for array indexes).
- Values do not have to be unique.
Java Map
Histogram of Words
reads file into a string (user function)

StringTokenizer is std class -- can be used to break up strings

If new word, add with count of 1, otherwise increment old value. Note use of Integer
public class Main {

    public static void main(String[] args) {
        // TODO code application logic here
        Map<String, Integer> histogram = new TreeMap<String, Integer>(); // (A)
        String allChars = ""; // must be defined outside try-catch
        if (args.length > 0) {
            try {
                allChars = getAllChars(args[0]); // (B)
            }
            catch (IOException ioe) {
                System.out.println("caught IOException");
                // exception handling in Chapter 10
            }
        }
        Stringtokenizer st = new Stringtokenizer(allChars); // (C)
        while (st.hasMoreTokens()) { // (D)
            String word = st.nextToken(); // (E)
            Integer count = (Integer) histogram.get(word); // (F)
            histogram.put(word, (count == null ? new Integer(1) : new Integer(count.intValue() + 1)));// (G)
        }
        System.out.println("Total number of DISTINCT words: " + histogram.size()); // (H)
        System.out.println(histogram); // (I)
    }

    static String getAllChars(String filename) throws IOException {

    }
}
read the file into a string
Class StringTokenizer

public class StringTokenizer
extends Object
implements Enumeration

The string tokenizer class allows an application to break a string into tokens. The tokenization method is much simpler than the one used by the StreamTokenizer class. The StringTokenizer methods do not distinguish among identifiers, numbers, and quoted strings, nor do they recognize and skip comments.

The set of delimiters (the characters that separate tokens) may be specified either at creation time or on a per-token basis.

An instance of StringTokenizer behaves in one of two ways, depending on whether it was created with the returnDelims flag having the value true or false:

- If the flag is false, delimiter characters serve to separate tokens. A token is a maximal sequence of consecutive characters that are not delimiters.
- If the flag is true, delimiter characters are themselves considered to be tokens. A token is thus either one delimiter character, or a maximal sequence of consecutive characters that are not delimiters.

A StringTokenizer object internally maintains a current position within the string to be tokenized. Some operations advance this current position past the characters processed.

A token is returned by taking a substring of the string that was used to create the StringTokenizer object.

The following is one example of the use of the tokenizer. The code:

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

<table>
<thead>
<tr>
<th>Type</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td><strong>countTokens()</strong></td>
<td>Calculates the number of times that this tokenizer's <code>nextToken</code> method can be called before it generates an exception.</td>
</tr>
<tr>
<td>boolean</td>
<td><strong>hasMoreElements()</strong></td>
<td>Returns the same value as the <code>hasMoreTokens</code> method.</td>
</tr>
<tr>
<td>boolean</td>
<td><strong>hasMoreTokens()</strong></td>
<td>Tests if there are more tokens available from this tokenizer's string.</td>
</tr>
<tr>
<td>Object</td>
<td><strong>nextElement()</strong></td>
<td>Returns the same value as the <code>nextToken</code> method, except that its declared return value is <code>Object</code> rather than <code>String</code>.</td>
</tr>
<tr>
<td>String</td>
<td><strong>nextToken()</strong></td>
<td>Returns the next token from this string tokenizer.</td>
</tr>
<tr>
<td>String</td>
<td><strong>nextToken(String delim)</strong></td>
<td>Returns the next token in this string tokenizer's string.</td>
</tr>
</tbody>
</table>
nextToken

public String nextToken(String delim)

Returns the next token in this string tokenizer's string. First, the set of characters considered to be delimiters by this StringTokenizer object is changed to be the characters in the string delim. Then the next token in the string after the current position is returned. The current position is advanced beyond the recognized token. The new delimiter set remains the default after this call.

Parameters:

delim - the new delimiters.

Returns:

the next token, after switching to the new delimiter set.

Throws:

NoSuchElementException - if there are no more tokens in this tokenizer's string.

Kind of like an array out of bounds exception
The constructor used in this program
Java generics - a latecomer

• All manipulations of generics and all type checking must be done at compile time.
  - This can lead to problems with arrays and assignments across generic types
  - Because all type checking is done at runtime, actual objects are compiled into an underlying base class
  - Thus, an array of Node<type1> and an array of Node<type2> will both be compiled into an array of Node that can hold either.
Erasure and reification

- Erasure is the process of throwing away type information after byte code is generated
  - List<T> simply becomes a list of objects rather than a list of objects of type T
  - This allowed Java generics to be backward compatible with existing VMs and existing code
Erasure and reification

reify |ˈrɛəfɪ| verb (-fies, -fied) [trans.] formal
make (something abstract) more concrete or real

• Reification, i.e. having generic types present at runtime as a concrete, testable entity, would break compatibility with previously existing code and/or the VM or require new APIs for collection classes to be written

• Two approaches to allow this ideal situation
  • Wait until all code is safe with current generics (e.g. no unchecked warnings), and then switch over VMs -- this will never happen
  • Carry runtime information along as extra parameters, etc. -- this will break pre-existing code
Practical implications of this

• For a type parameter $T$, you can't write a class literal $T.class$.
• You cannot use $instanceof$ to test if an object is of type parameter $T$.
• You cannot create an array of type parameter $T$.
• You cannot write class literals for generic types like $List<String>.class$. 
Practical implications of this

Cannot test if an object is an `instanceof List<String>`, or create an array of `List<String>`.

- Leads to unchecked operations -- operations that normally would cause runtime checks but cannot do so because not enough information is available. For example, a cast to the type `List<String>` is an unchecked cast, because the generated code checks that the object is a List but doesn't check whether it is the right kind of
What is not an implication of this

• Java is still safe
• Why?
  – Runtime checks can still decide if a particular field of the Foo<T> is an actual type (as opposed to a type parameter) of T or something else
  – If it is something else, a ClassCastException
  – In the example below, `ls.get(0)` returns a `Number` for 0 and not a string, and the runtime check will catch this.

```java
List ln = new ArrayList<Number>();
List<String> ls = ln; // unchecked warning
String s = ls.get(0); // ClassCastException
```