

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

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

```
class myApp {  
    GPSUser handler = ...  
        new appGPSHandler( )  
    ...  
    GPS.registerAlarm(handler);  
    ...  
}
```

- 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

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(...);  
            }  
        }  
    }
```

}
YHL/SPM

- 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

Container Class

4

```

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

```

```

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

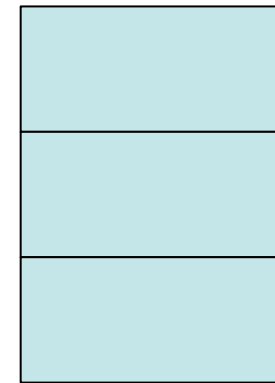
```

```

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

```

List *l*



```

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

```

```

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

```

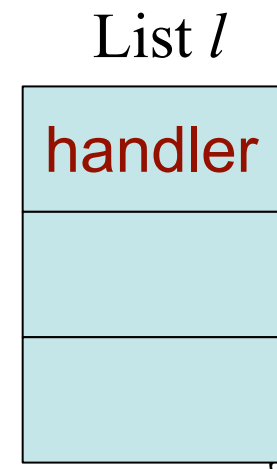
YHL/SPM

Container Class

```

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

```



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

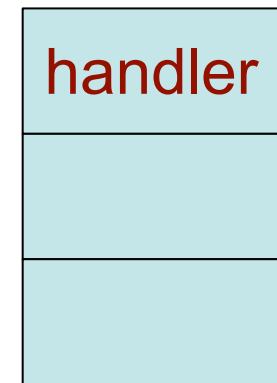
```
class myApp {
    . . .
    GPS.register(handler);
    . . .
}
```

YHL/SPM

Container Class

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

List *l*



```

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

```

```

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

```

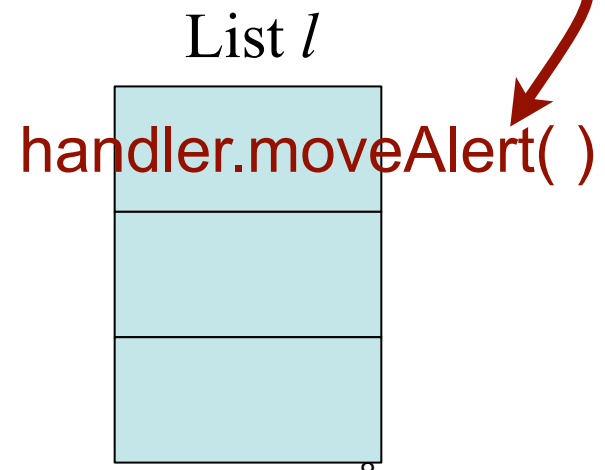
YHL/SPM

Container Class

```

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

```




```

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 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 → address, student ID → name, course title → 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

operation	vector	deque	list
array-like access	$O(1)$	$O(1)$	$O(N)$
insert/delete at front	$O(N)$	$O(1)+$	$O(1)$
insert/delete at end	$O(1)+$	$O(1)+$	$O(1)$
insert/delete in middle	$O(N)$	$O(N)$	$O(1)$

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

List (Java Platform SE 6) - Mozilla Firefox

File Edit View History Bookmarks Yahoo! Tools Help

http://java.sun.com/javase/6/docs/api/java/util/List.html

Overview Package **Class** Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS

SUMMARY: NESTED | FIELD | CONSTR | METHOD

DETAIL: FIELD | CONSTR | METHOD

Java™ Platform
Standard Ed. 6

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 conceivable that

Done

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*.

List (Java Platform SE 6) - Mozilla Firefox

File Edit View History Bookmarks Yahoo! Tools Help

http://java.sun.com/javase/6/docs/api/java/util/List.html

Method Summary

boolean	<code>add(E e)</code> Appends the specified element to the end of this list (optional operation).
void	<code>add(int index, E element)</code> Inserts the specified element at the specified position in this list (optional operation).
boolean	<code>addAll(Collection<? extends E> c)</code> Appends 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).
boolean	<code>addAll(int index, Collection<? extends E> c)</code> Inserts all of the elements in the specified collection into this list at the specified position (optional operation).
void	<code>clear()</code> Removes all of the elements from this list (optional operation).
boolean	<code>contains(Object o)</code> Returns true if this list contains the specified element.
boolean	<code>containsAll(Collection<?> c)</code> Returns true if this list contains all of the elements of the specified collection.
boolean	<code>equals(Object o)</code> Compares the specified object with this list for equality.

Done

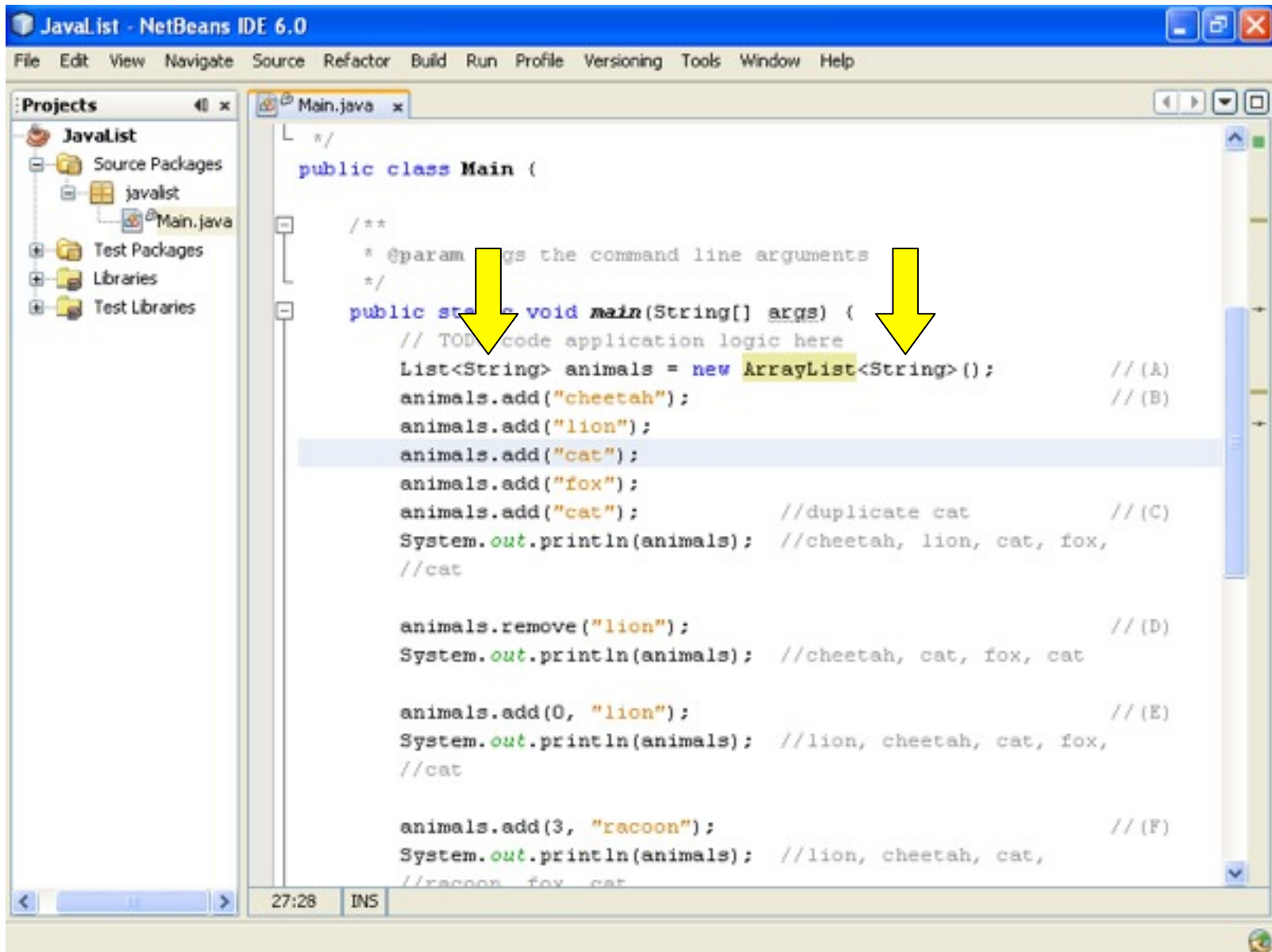
List (Java Platform SE 6) - Mozilla Firefox

File Edit View History Bookmarks Yahoo! Tools Help

http://java.sun.com/javase/6/docs/api/java/util/List.html

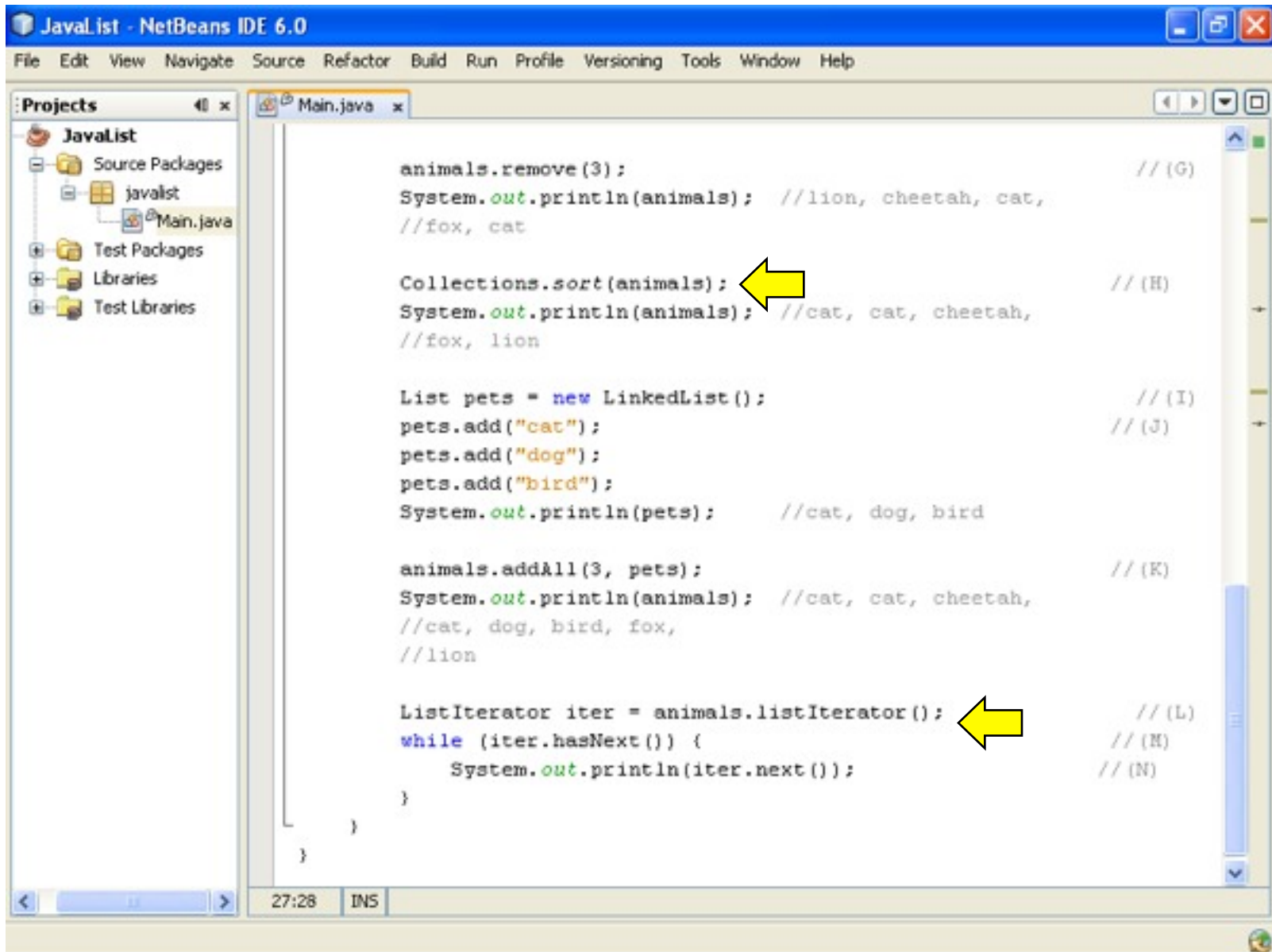
ListIterator <E>	listIterator () Returns a list iterator over the elements in this list (in proper sequence).
ListIterator <E>	listIterator (int index) Returns a list iterator of the elements in this list (in proper sequence), starting at the specified position in this list.
E	remove (int index) Removes the element at the specified position in this list (optional operation).
boolean	remove (Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
boolean	removeAll (Collection <?> c) Removes from this list all of its elements that are contained in the specified collection (optional operation).
boolean	retainAll (Collection <?> c) Retains only the elements in this list that are contained in the specified collection (optional operation).
E	set (int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	size () Returns the number of elements in this list.

Done



What is an array list?

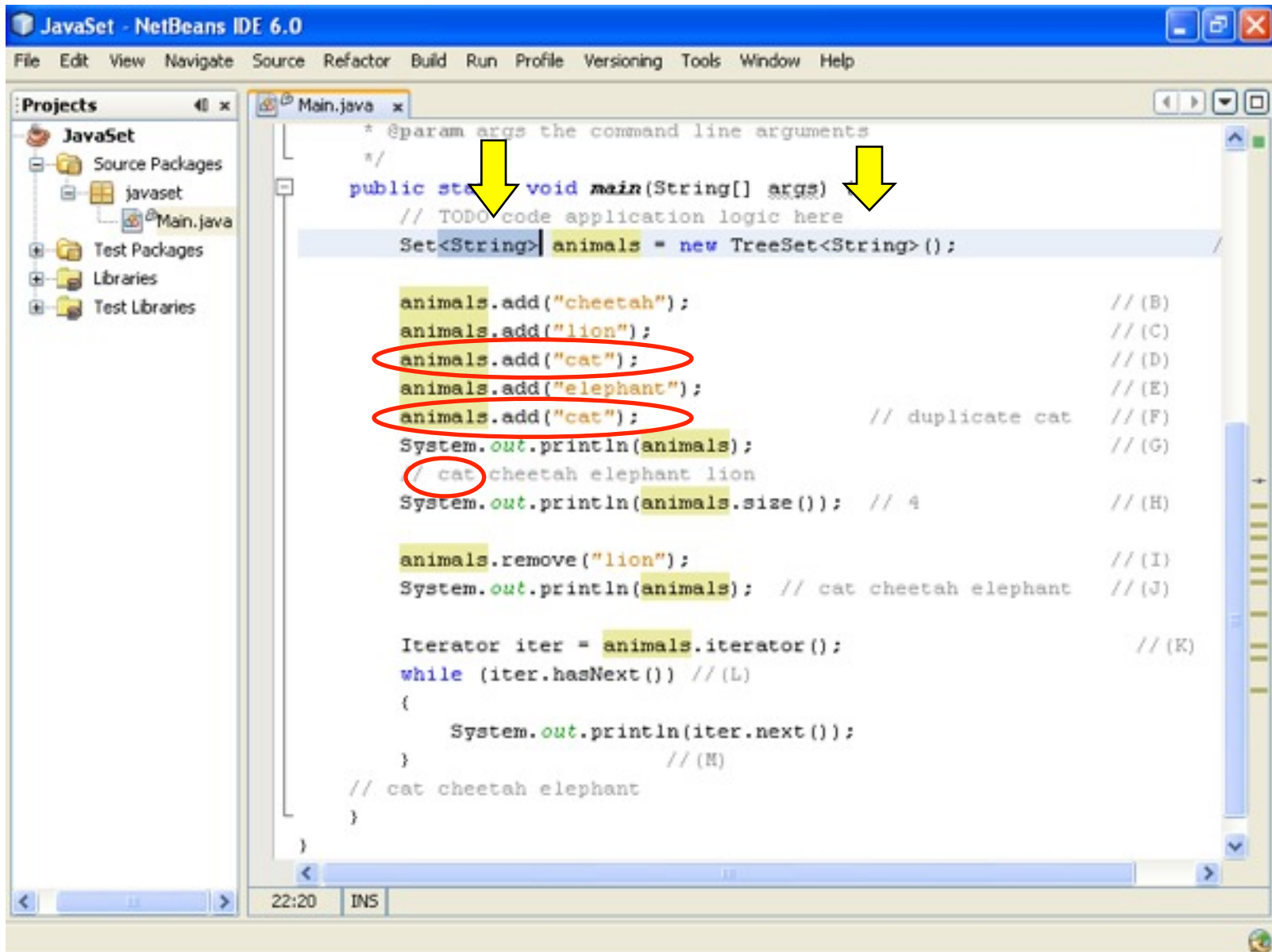
- Is it an an Array or is it a list?
- From <http://docs.oracle.com/javase/6/docs/api/java/util/ArrayList.html>
 - Resizable-array implementation of the `List` interface. Implements all optional list operations, and permits all elements, including `null`.
 - In addition to implementing the `List` interface, 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.)



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



All Implemented Interfaces:[Cloneable](#), [Collection](#), [Serializable](#), [Set](#), [SortedSet](#)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](#)

JavaSet - NetBeans IDE 6.0

File Edit View Navigate Source Refactor Build Run Profile Versioning Tools Window Help

Projects

- JavaSet
 - Source Packages
 - javaset
 - Main.java
 - Test Packages
 - Libraries
 - Test Libraries

Main.java

```
// TODO code application logic here
Set<String> animals = new TreeSet<String>();

animals.add("cheetah"); // (B)
animals.add("lion"); // (C)
animals.add("cat"); // (D)
animals.add("elephant"); // (E)
animals.add("cat"); // duplicate cat // (F)
System.out.println(animals); // (G)
// cat cheetah elephant lion
System.out.println(animals.size()); // 4 // (H)

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

Iterator iter = animals.iterator(); // (K)
while (iter.hasNext()) // (L)
{
```

Output

JavaSet (run-single) x :ext:ee462b30@msee190pc3.ecn.purdue.edu:/home/shay/a/ee462b30/projects x

run-single:
[cat, cheetah, elephant, lion]
4
[cat, cheetah, elephant]
cat

no duplicate element in Set
elements sorted

Java Map

Map (Hash Table)

- array: integer \rightarrow element (object)
- map: key (object, integer, or string ...) \rightarrow value (object)
- example:
 - name \rightarrow phone number
 - student ID \rightarrow department
 - city name \rightarrow 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

JavaMap - NetBeans IDE 6.0

File Edit View Navigate Source Refactor Build Run Profile Versioning Tools Window Help

Main.java x

```
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]);  
            } catch (IOException ioe) {  
                System.out.println("caught IOException");  
                // exception handling in Chapter 10  
            }  
            StringTokenizer st = new StringTokenizer(allChars);  
            while (st.hasMoreTokens()) {  
                String word = st.nextToken();  
                Integer count = (Integer) histogram.get(word);  
                histogram.put(word, (count == null ? new Integer(1)  
                    : new Integer(count.intValue() + 1)));  
            }  
            System.out.println("Total number of DISTINCT words: " + histogram.size());  
            System.out.println(histogram);  
        }  
    }  
    static String getAllChars(String filename) throws IOException {  
        // ...  
    }  
}
```

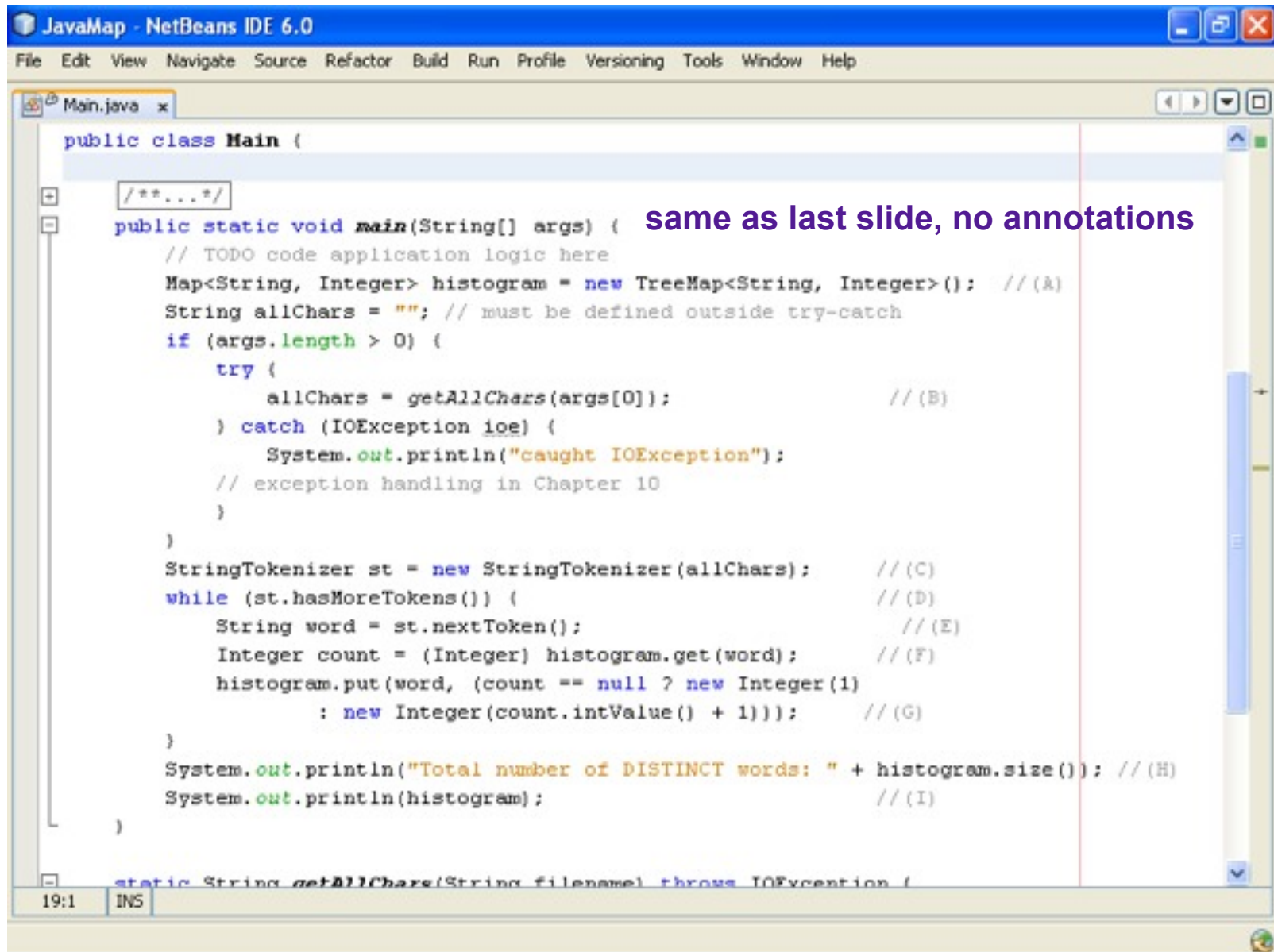
reads file into a string
(user function)

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

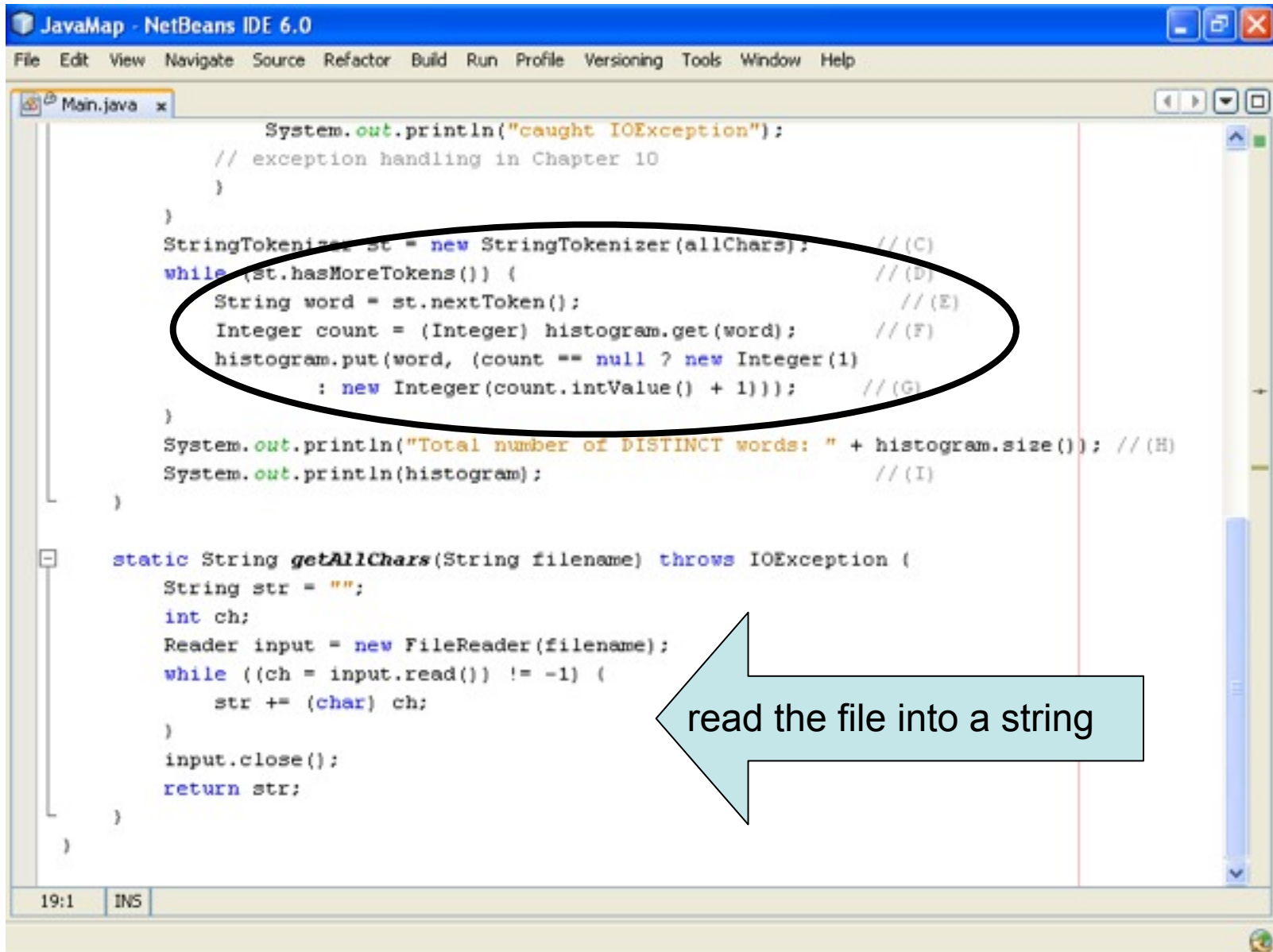
19:1 INS

The last slide without text



```
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 {
```

19:1 INS



```
JavaMap - NetBeans IDE 6.0
File Edit View Navigate Source Refactor Build Run Profile Versioning Tools Window Help

Main.java x
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 {
    String str = "";
    int ch;
    Reader input = new FileReader(filename);
    while ((ch = input.read()) != -1) {
        str += (char) ch;
    }
    input.close();
    return str;
}
}
```

19:1 INS

read the file into a string

java.util

Class StringTokenizer

[java.lang.Object](#)

↳ [java.util.StringTokenizer](#)

All Implemented Interfaces:

[Enumeration](#)

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:

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

Method Summary

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

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:

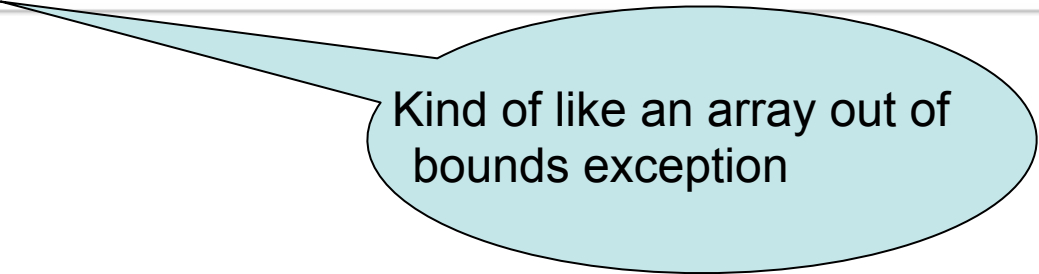
delim - the new delimiters.

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

StringTokenizer

```
public StringTokenizer(String str,  
                      String delim,  
                      boolean returnDelims)
```

Constructs a string tokenizer for the specified string. All characters in the `delim` argument are the delimiters for separating tokens.

If the `returnDelims` flag is `true`, then the delimiter characters are also returned as tokens. Each delimiter is returned as a string of length one. If the flag is `false`, the delimiter characters are skipped and only serve as separators between tokens.

Note that if `delim` is null, this constructor does not throw an exception. However, trying to invoke other methods on the resulting `StringTokenizer` may result in a `NullPointerException`.

Parameters:

`str` - a string to be parsed.
`delim` - the delimiters.
`returnDelims` - flag indicating whether to return the delimiters as tokens.

StringTokenizer

```
public StringTokenizer(String str,  
                      String delim)
```

Constructs a string tokenizer for the specified string. The characters in the `delim` argument are the delimiters for separating tokens. Delimiter characters themselves will not be treated as tokens.

Parameters:

`str` - a string to be parsed.
`delim` - the delimiters.

The constructor used in this program

StringTokenizer

```
public StringTokenizer(String str)
```

Constructs a string tokenizer for the specified string. The tokenizer uses the default delimiter set, which is `"\n\t\r\f"`: the space character, the tab character, the newline character, the carriage-return character, and the form-feed character. Delimiter characters themselves will not be treated as tokens.

Parameters:

`str` - a string to be parsed.

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 object of type T
 - This allowed Java generics to be backward compatible with existing VMs and existing code

Erasure and reification

reify |'rēəfī|

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 *instance of* `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.

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

see <http://www.angelikalanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html#FAQ001>

YHL/SPM

and <http://gafter.blogspot.com/2006/11/reified-generics-for-java.html> among other sources

Container, Class

47