LOYOLA
UNIVERSITY
CHICAGO

COMP 313 / 488: Intermediate Object-Oriented Programming

Dr. Robert Yacobellis
Advanced Lecturer
Department of Computer Science

Week 4 Supplemental Topics

- **Java Collections**
- Java data structures
- Object-inherited methods

Collections

- A *collection* — sometimes called a container — is simply an object that groups multiple elements into a single unit. Collections are used to store, retrieve, manipulate, and communicate aggregate data.

- A *collections framework* is a unified architecture for representing and manipulating collections. All collections frameworks contain:

  - **Interfaces:** abstract data types that represent collections; allow collections to be manipulated independently of their representation details

  - **Implementations:** reusable concrete implementations of collection interfaces

  - **Algorithms:** reusable, polymorphic methods that perform useful computations like searching and sorting on objects that implement collection interfaces

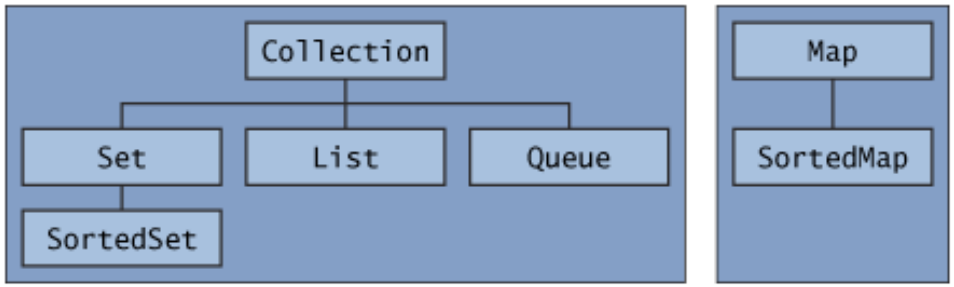
Java Collections Framework Benefits

- Reduces programming effort
- Increases program speed and quality
- Reduces the effort to learn and use new APIs based on standard collection interfaces
- Fosters software reuse via conformance to standard collection interfaces

LOYOLA
UNIVERSITY
CHICAGO

Java Core Collections Interfaces

The *core collection interfaces* encapsulate different types of collections, which are shown in the figure below. These interfaces allow collections to be manipulated independently of the details of their representation. Core collection interfaces are the foundation of the Java Collections Framework. As you can see in the following figure, the core collection interfaces form a hierarchy.



The core collection interfaces.

A `Set` is a special kind of `Collection`, a `SortedSet` is a special kind of `Set`, and so forth. Note also that the hierarchy consists of two distinct trees — a `Map` is not a true `Collection`.

Note that all the core collection interfaces are generic. For example, this is the declaration of the `Collection` interface.

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

Example: Set<Integer> intSet = new TreeSet<Integer>(…);

The `<E>` syntax tells you that the interface is generic. When you declare a `Collection` instance you can *and should* specify the type of object contained in the collection. Specifying the type allows the compiler to verify (at compile-time) that the type of object you put into the collection is correct, thus reducing errors at runtime.

http://java.sun.com/docs/books/tutorial/collections/interfaces/

Core Collections Interfaces

- Collection – the root interface of the Collections hierarchy
  - Set – a collection that does not contain duplicates
    - SortedSet – an always-sorted collection of non-duplicated objects
    - Implemented by **HashSet** (unsorted) and **TreeSet** (sorted)
  - List – an ordered collection that can contain duplicates
    - Implemented by **ArrayList** and **LinkedList**
  - Queue – typically, a first-in, first-out (FIFO) collection that models a waiting line; other orders can be specified (LIFO, …)
    - Implemented by **LinkedList** and **PriorityQueue**, among others
- Map – the root of the Map hierarchy; Maps associate unique keys to values
  - SortedMap – maintains its keys in sorted order
  - Implemented by **HashMap** (unsorted) and **TreeMap** (sorted)

The Collection Interface

- **All Collection implementations have a constructor that takes a Collection argument and imports that argument's elements into the Collection (the *conversion constructor*).**
- Standard Collection operations (methods):

```
public interface Collection<E> extends Iterable<E> { // every Collection is also
    Iterable
  // Basic Collection operations
  int size();
  boolean isEmpty(), contains(Object element);
  boolean add(E element), remove(Object element); // optional
  Iterator<E> iterator(); // required because of extending the Iterable interface
  // Bulk Collection operations
  boolean containsAll(Collection<?> c);
  boolean addAll(Collection<? extends E> c); // optional
  boolean removeAll(Collection<?> c), retainAll(Collection<?> c); // optional
  void clear(); // optional
  // Array operations
  Object[] toArray();
  <T> T[] toArray(T[] a);
}
```

Traversing Collections

There are two ways to traverse collections: (1) with the `for-each` construct and (2) by using `Iterator`s.

for-each Construct

The `for-each` construct allows you to concisely traverse a collection or array using a `for` loop — see The for Statement. The following code uses the `for-each` construct to print out each element of a collection on a separate line.

```
for (Object o : collection)
    System.out.println(o);   /// implicitly uses each collection element's
                             toString() method
```

Iterators

An `Iterator` is an object that enables you to traverse through a collection and to remove elements from the collection selectively, if desired. You get an `Iterator` for a collection by calling its `iterator` method. The following is the `Iterator` interface.

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

To traverse the collection and remove specific elements:

```
static void filter(Collection<?> c) {
    for (Iterator<?> it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

The `hasNext` method returns `true` if the iteration has more elements, and the `next` method returns the next element in the iteration. The `remove` method removes the last element that was returned by `next` from the underlying `Collection`. The `remove` method may be called only once per call to `next` and throws an exception if this rule is violated.

Note that `Iterator.remove` is the *only* safe way to modify a collection during iteration; the behavior is unspecified if the underlying collection is modified in any other way while the iteration is in progress.

http://java.sun.com/docs/books/tutorial/collections/interfaces/

The Collections Class

- Class **Collections** provides static methods that search, sort, and provide other operations on collections

Method	Description
`sort`	Sorts the elements of a `List`.
`binarySearch`	Locates an object in a `List`.
`reverse`	Reverses the elements of a `List`.
`shuffle`	Randomly orders a `List`'s elements.
`fill`	Sets every `List` element to refer to a specified object.
`copy`	Copies references from one `List` into another.
`min`	Returns the smallest element in a `Collection`.
`max`	Returns the largest element in a `Collection`.
`addAll`	Appends all elements in an array to a `Collection`.
`frequency`	Calculates how many collection elements are equal to the specified element.
`disjoint`	Determines whether two collections have no elements in common.

Lists have only these

Source: Deitel & Deitel: Java – How to Program

The Collections Class – 2

- Class **Collections** also provides **wrapper methods** that enable synchronization and unmodifiability

public static method headers - Synchronization wrapper methods

```
< T > Collection< T > synchronizedCollection( Collection< T > c )
< T > List< T > synchronizedList( List< T > aList )
< T > Set< T > synchronizedSet( Set< T > s )
< T > SortedSet< T > synchronizedSortedSet( SortedSet< T > s )
< K, V > Map< K, V > synchronizedMap( Map< K, V > m )
< K, V > SortedMap< K, V > synchronizedSortedMap( SortedMap< K, V > m )
```

public static method headers - Unmodifiability wrapper methods

```
< T > Collection< T > unmodifiableCollection( Collection< T > c )
< T > List< T > unmodifiableList( List< T > aList )
< T > Set< T > unmodifiableSet( Set< T > s )
< T > SortedSet< T > unmodifiableSortedSet( SortedSet< T > s )
< K, V > Map< K, V > unmodifiableMap( Map< K, V > m )
< K, V > SortedMap< K, V > unmodifiableSortedMap( SortedMap< K, V > m )
```

Source: Deitel & Deitel: Java – How to Program

Abstract Collection Implementations

- The Java collections framework also provides abstract implementations of various Collection interfaces:

AbstractCollection<E>
AbstractSet<E>
AbstractList<E>, AbstractSequentialList<E>
AbstractQueue<E>
AbstractMap<K,V>
AbstractMap.SimpleEntry<K,V>
AbstractMap.SimpleImmutableEntry<K,V>

General-Purpose Collections Framework Implementations

The general-purpose implementations are summarized in the following table.

General-purpose Implementations

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
`Set`	`HashSet`		`TreeSet`		`LinkedHashSet`
`List`		`ArrayList`		`LinkedList`	
`Queue`					
`Map`	`HashMap`		`TreeMap`		`LinkedHashMap`

The Set<E> Interface

- Corresponds to the mathematical definition of a set (no duplicates are allowed)
- Compared to the general Collection<E> interface:
  - Interface is identical
  - Every constructor must create a collection without duplicates
  - The operation add cannot add an element already in the set
    - Returns true if a new element was added to the set
  - The method call set1.equals(set2) works as follows:
    - True if and only if set1 Í set2 and set2 Í set1
- Set idioms:
  - set1.addAll(set2) is set1 È set2 (set union)
  - set1.retainAll(set2) is set1 Ç set2 (set intersection)
  - set1.removeAll(set2) is set1 - set2 (set subtraction)

HashSet and TreeSet Classes

- HashSet<E> and TreeSet<E> implement the Set<E> interface

- HashSet<E> (and LinkedHashSet<E>)
  - Implemented using a hash table (and the hashCode() method)
  - No ordering of elements (LinkedHashSet elements are ordered)
  - Add, remove, and contains have constant time complexity *O(c)*

- TreeSet<E>
  - Implemented using a tree structure
  - Guarantees ordering of elements based on comparison vs. insertion
  - Add, remove, and contains methods have logarithmic time complexity *O(log(n))*, where *n* is the number of elements in the set

HashSet Example (pre-generics)

```
// [Source: java.sun.com]
import java.util.*;
public class FindDups {
   public static void main(String args[]) { // inputs are command line words
      Set s = new HashSet(); // upcasting using the Set interface
      for (int i = 0; i < args.length; i++) {
         if (!s.add(args[i])) // note that s can hold any type of Object
            System.out.println("Duplicate detected: " + args[i]);
      }
      System.out.println(s.size() + " distinct words detected: " + s);
   }
}
// Note: s prints out as [word, word, word, …] with arbitrary word ordering
```

For an example of using TreeSet:
http://www.idevelopment.info/data/Programming/java/collections/TreeSetExample.java

The List<E> Interface

- Corresponds to an ordered group of elements; duplicates allowed
- Extensions compared to the Collection<E> interface:
  - Access to elements via indexes, like arrays
    - add(int, E) (add at this index), get(int), set(int, E) (in-place update)
    - E remove(int) (remove element at this index)
  - Search for elements
    - indexOf(Object), lastIndexOf(Object) (rightmost occurrence)
  - Specialized Iterator<E>, called ListIterator<E>
  - Extraction of sublist
    - subList(int fromIndex, int toIndex)
- list1.equals(list2) takes element ordering into consideration
  - list1.equals(list2) → list1.hashCode() == list2.hashCode()

ArrayList<E> and LinkedList<E> Classes

- These implement the List<E> interface

- ArrayList<E> is an array-based implementation where elements can be accessed directly by index via the get() and set() methods
  - Default choice for a simple sequence
  - Backed internally by an array that is grown & copied when necessary
  - Like an "array on steroids" with no predefined limit on # elements

- LinkedList<E> is based on a doubly-linked list
  - Better performance on add() and remove() compared to ArrayList<E>, but poorer performance on get() and set() methods

ArrayList Example (pre-generics)

```
// [Source: java.sun.com]
import java.util.*;
public class Shuffle {
   public static void main(String args[]) {
      List al = new ArrayList();
      for (int i = 0; i < args.length; i++)
         al.add(args[i]); // again, al can hold any type of Object
      Collections.shuffle(al, new Random()); // static method
      System.out.println(al);
   }
}
```

For more:
http://www.idevelopment.info/data/Programming/java/collections/ArrayListExample.java

LinkedList Example (pre-generics)

```
import java.util.*;
public class MyStack { // implemented via LinkedList; should now be MyStack<E>
   private LinkedList list = new LinkedList();    // should be LinkedList<E> list = …
   public void push(Object o) { list.addFirst(o); }    // should be type E, not Object
   public Object top() { return list.getFirst(); }        // should return type E vs Object
   public Object pop() { return list.removeFirst(); } // should return type E

   public static void main(String args[]) {
      Car myCar; // some random Car class
      MyStack s = new MyStack();              // should be MyStack<Car> s = …
      s.push(new Car());
      myCar = (Car) s.pop();             // cast not needed if using generics
   }
}
```

For more:
http://www.idevelopment.info/data/Programming/java/collections/LinkedListExample.java

How to Remove List Duplicates

```
import java.util.*;
public class NoDups {
   private List<String> list;

   public static void main(String args[]) {
      list = new LinkedList<String>();
      addElements(list); // some method that fills the list with elements
      list = new LinkedList<String>(new LinkedHashSet<String>(list));
      // removes duplicates – Sets don't allow them
      // depends on conversion constructors; maintains insertion ordering
      // if ordering is not important, use new HashSet<String>(list) instead
      // this same approach will also work with an ArrayList
   }
}
```

The Map<K,V> Interface

- Maps keys to values; also called an *associative array* or *dictionary*
- Methods for adding and deleting
  - put(K key, V value) // returns previous value of type V, or null
  - remove(Object key) // returns previous value of type V, or null
- Methods for extracting objects
  - get(Object key) // returns a value of type V, or null if no entry
- Methods to retrieve the keys, values, and (key, value) pairs
  - keySet() // returns a Set<K> (keys can't be duplicated)
  - values() // returns a Collection<V> (values can be)
  - entrySet() // returns a Set<Map.Entry<K,V>>
  - Note: Map.Entry is a nested static interface within Map

HashMap<K,V> and TreeMap<K,V> Classes

- HashMap<K,V> and TreeMap<K,V> implement the Map<K,V> interface

- HashMap<K,V> (and LinkedHashMap<K,V>)
  - Implemented using a hash table
  - No ordering of (key, value) pairs
  - LinkedHashMap (key, value) pairs are ordered based on insertion

- TreeMap<K,V>
  - Implemented using a “red-black” tree structure
  - (key, value) pairs are ordered on the key

HashMap Example (pre-generics)

```
import java.util.*;
   public class Freq {
      private static final Integer ONE = new Integer(1); // or "= 1;" w/autoboxing
      public static void main(String args[]) {
         Map m = new HashMap(); // should be Map<Integer> m = …
         // Initialize frequency table from command line
         for (int i=0; i < args.length; i++) {
            Integer freq = (Integer) m.get(args[i]); // no cast if generic
            m.put(args[i], (freq == null ? ONE :
                        new Integer(freq.intValue() + 1))); // *
         } // Note: the Integer manipulations can be done via autoboxing
         System.out.println(m.size() + " distinct words detected:");
         System.out.println(m);
      }
   }       // * Who can tell me how this Java construct (x ? y : z) works?
```

TreeMap<K,V> Example

```
import java.util.*;
    public class TreeMapExample{
      public static void main(String[] args) {
        System.out.println("TreeMap Example!\n");
        TreeMap <Integer, String> tMap = new TreeMap<Integer, String>(); // type-specific TreeMap
        tMap.put(1, "Sunday"); tMap.put(2, "Monday");  tMap.put(3, "Tuesday"); tMap.put(4, "Wednesday");
        tMap.put(5, "Thursday"); tMap.put(6, "Friday"); tMap.put(7, "Saturday"); // install values with Integer keys 1-7

        System.out.println("Keys of tree map: " + tMap.keySet());    // retrieving all keys
        System.out.println("Values of tree map: " + tMap.values());    // retrieving all values
        System.out.println("Key: 5 Value: " + tMap.get(5) + "\n");                                    //
    retrieving the value from key with key number 5
        System.out.println("First key: " + tMap.firstKey() + " Value: "    // retrieving the First key and its value
                                  + tMap.get(tMap.firstKey()) + "\n");
        System.out.println("Last key: " + tMap.lastKey() + " Value: "     // retrieving the Last key and value
                                    + tMap.get(tMap.lastKey()) + "\n");

        System.out.println("Removing first value: " + tMap.remove(tMap.firstKey())); // removing the first key and value
        System.out.println("Now the tree map Keys are: " + tMap.keySet());
        System.out.println("Now the tree map contains: " + tMap.values() + "\n");

        System.out.println("Removing last value: " + tMap.remove(tMap.lastKey())); // removing the last key and value
        System.out.println("Now the tree map Keys are: " + tMap.keySet());
        System.out.println("Now the tree map contains: " + tMap.values());
      }
    }
```

LOYOLA UNIVERSITY CHICAGO

TreeMap<K,V> Example Output

TreeMap Example!

Keys of tree map: [1, 2, 3, 4, 5, 6, 7]
Values of tree map: [Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]
Key: 5 Value: Thursday

First key: 1 Value: Sunday

Last key: 7 Value: Saturday

Removing first value: Sunday
Now the tree map Keys are: [2, 3, 4, 5, 6, 7]
Now the tree map contains: [Monday, Tuesday, Wednesday, Thursday, Friday, Saturday]

Removing last value: Saturday
Now the tree map Keys are: [2, 3, 4, 5, 6]
Now the tree map contains: [Monday, Tuesday, Wednesday, Thursday, Friday]

Quick Informal Quiz

Given:

```
1.  import java.util.*;
2.  public class Average {
3.      static LinkedList<Double> myList;
4.      public static double mean() {
5.          double sum = 0;
6.          for (Double element : myList)
7.              sum += element;
8.          return sum / myList.size();
9.      }
10.     public static void main(String[] args) {
11.         LinkedList<Double> myList = new LinkedList<Double>();
12.         myList.add(2.0); myList.add(4.0);
13.         System.out.println(mean());
14.     }
15. } // end class Average
```

This myList is initialized to null and never set

A NullPointerException is thrown here → it references the null static variable myList

This is a local variable, so the static variable myList is not changed!

What is the result (Choose all that apply.)

A. Compilation succeeds

B. Compilation fails with an error on line 11

C. Compilation fails with an error on line 12

D. Compilation fails with an error on line 13

E. The program runs with output 3

F. The program runs with output 2

G. An exception is thrown at runtime

Why??

What happens if line 11 is fixed??

Week 4 Supplemental Topics

- Java Collections
- **Java data structures**
  - **HashSets, etc.**
  - Arrays
  - Trees & tree processing
- Object-inherited methods

HashSet<E>

- Remember that HashSet<E> extends AbstractSet<E> and implements the Set<E> interface (among others)

- HashSet<E>
  - Implemented using a hash table (a backing HashMap structure)
  - No ordering of elements
  - Add, remove, and contains methods have constant time complexity *O(c)*

- Common methods include add(), size(), remove(), clear()

The Four HashSet Constructors

- The first form constructs a default hash set with initial capacity 16 and load factor 0.75 (see below for more discussion of these items):
  - HashSet( )
- The following constructor form initializes the hash set by using the elements of c.
  - HashSet(Collection<? extends E> c) – this is the standard conversion constructor
  - The loadFactor is 0.75 and the initial capacity is large enough to contain the Collection elements.
- The following constructor form initializes the capacity of the hash set to initialCapacity. The capacity grows automatically as elements are added to the HashSet.
  - HashSet(int initialCapacity)
  - **Reasonable approach: set the capacity to about twice the number of expected elements – a HashSet always maintains a number of hash buckets that is a power of two, and its capacity doubles each time it is expanded**
- The fourth form initializes both the capacity and the fill ratio (also called load capacity) of the hash set from its arguments:
  - HashSet(int initialCapacity, float loadFactor)
  - Here the loadFactor must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its loadFactor, the hash set is expanded.

Some Common HashSet<E> Methods

boolean add(E o)
Adds the specified element to this set if it is not already present. Returns true if the element was added.

void clear()
Removes all of the elements from this set.

Object clone()
Returns a shallow copy of this HashSet instance; the elements themselves are not cloned.

boolean contains(Object o)
Returns true if this set contains the specified element o.

boolean isEmpty()
Returns true if this set contains no elements.

Iterator<E> iterator()
Returns an iterator over the elements in this set.

boolean remove(Object o)
Removes the specified element from this set and returns true if it is present.

int size()
Returns the number of elements in this set (its cardinality).

A Simple HashSet<E> Example

```
import java.util.*;

public class CollectionTest {
 public static void main(String [] args) {
  System.out.println( "Collection Example!\n" );
  int size;
  // Create a collection
  HashSet<String>collection =
     new HashSet<String>();
  String str1 = "Yellow", str2 = "White",
     str3 = "Green", str4 = "Blue";
  //Adding data in the collection
  collection.add(str1);
  collection.add(str2);
  collection.add(str3);
  collection.add(str4);
  System.out.print("Collection data: ");
  //Create an iterator
  Iterator iterator = collection.iterator();
  while (iterator.hasNext()){ // random ordering
    System.out.print(iterator.next() + " ");
  }
  System.out.println(); // continued at right …
  size = collection.size(); // Get size of the collection
  if (collection.isEmpty())
    System.out.println("Collection is empty");
  else
    System.out.println( "Collection size: " + size);
  System.out.println();
  // Remove specific data
  collection.remove(str2);
  System.out.println("After removing [" + str2 + "]\n");
  System.out.print("Now collection data is: ");
  iterator = collection.iterator(); // resets the iterator
  while (iterator.hasNext()) {
    System.out.print(iterator.next() + " ");
  }
  System.out.println();
  size = collection.size();
  System.out.println("Collection size: " + size + "\n");
  // empty the collection
  collection.clear();
  size = collection.size();
  if (collection.isEmpty())
    System.out.println("Collection is empty");
  else
    System.out.println( "Collection size: " + size);
 }
}
```

Program Output:

```
javac CollectionTest.java
java CollectionTest
Collection Example!

Collection data: Blue White Green Yellow
Collection size: 4

After removing [White]

Now collection data is: Blue Green Yellow
Collection size: 3

Collection is empty
```

Source: http://www.roseindia.net/java/jdk6/CollectionTest.shtml

Finding the Minimum Element in a HashSet (pre-generics example)

```
/*
 Find Minimum element of Java HashSet Example
 This java example shows how to find a minimum
 element of Java HashSet using the min() method
 of the Collections class.
*/
import java.util.HashSet;
import java.util.Collections;
public class FindMinimumOfHashSetExample {
 public static void main(String[] args) {
  //create a HashSet object
  HashSet hashSet = new HashSet();
  //Add elements to HashSet
  hashSet.add(new Long("923740927"));
  // Note: can use generics + autoboxing after Java 5.0 …
  hashSet.add(new Long("4298748382"));
  hashSet.add(new Long("2374324832"));
  hashSet.add(new Long("2473483643"));
  hashSet.add(new Long("32987432984"));
/*
   To find the minimum element of Java HashSet, can use the
   static Object min(Collection c) method of the Collections class*.

   This method returns the minimum element of the HashSet according to
   its natural ordering.
  */
  Object obj = Collections.min(hashSet);
  System.out.println("Minimum Element of Java HashSet is : " + obj);
 }
}
/*
Output would be
Minimum Element of Java HashSet is : 923740927
*/
```

*** After Java 5.0, the genericized definition of min is much more complicated:**

```
static  <T extends Object & Comparable<? super T>>
        T min(Collection<? extends T> coll)
```

Source: http://www.java-examples.com/find-minimum-element-java-hashset-example

HashSet vs. LinkedHashSet, TreeSet

```
import java.util.*;

  public class SetDemo1 {
     static final int MIN = 1;
     static final int MAX = 10;

     public static void main(String args[]) {
        Set sethash = new HashSet();
        for (int i = MAX; i >= MIN; i--) {
           sethash.add(new Integer(i*i));
        } // adds in squares of numbers 1-10
        System.out.println("HashSet = " + sethash);

        Set setlink = new LinkedHashSet();
        for (int i = MAX; i >= MIN; i--) {
           setlink.add(new Integer(i*i));
        }
        System.out.println(
                  "LinkedHashSet = " + setlink);

        Set settree = new TreeSet();
        for (int i = MAX; i >= MIN; i--) {
           settree.add(new Integer(i*i));
        }
        System.out.println("TreeSet = " + settree);
     }
  }
```

Program Output:

HashSet = [9, 25, 4, 36, 100, 1, 49, 81, 16, 64] apparently random order

LinkedHashSet = [100, 81, 64, 49, 36, 25, 16, 9, 4, 1] insertion order

TreeSet = [1, 4, 9, 16, 25, 36, 49, 64, 81, 100] ascending sorted order
(Integers are Comparable)

Source: http://java.sun.com/developer/JDCTechTips/2002/tt1105.html

Cost Issues for HashSet, LinkedHashSet, and TreeSet

```
import java.util.*;

public class SetDemo2 {
   static final int N = 500000;
   static List numlist = new ArrayList(N);
   // fill List with 500,000 random Integer values
   static {
      Random rn = new Random();
      for (int i = 0; i < N; i++) {
         numlist.add(new Integer(rn.nextInt()));

      }
   }

   static void dotime(Set set, String which) {
      long start = System.currentTimeMillis();
      // add all List values to Set
      set.addAll(numlist);
      // look up all List values in Set
      for (int i = 0; i < N; i++) {
         if (!set.contains(numlist.get(i))) {
            System.out.println("contains error");

         }
      }
      long elapsed = System.currentTimeMillis() - start;

      System.out.println(which + " " + elapsed);
   } // continued at right …
```

```
   public static void main(String args[]) {
      if (args.length != 1) {
         System.out.println(
            "Usage: java SetDemo2 " +
            "HashSet|LinkedHashSet|TreeSet");
         System.exit(1);
      }

      // do timing of HashSet /
      // LinkedHashSet / TreeSet

      if (args[0].equals("HashSet"))
         dotime(new HashSet(), "HashSet");
      else if (args[0].equals("LinkedHashSet"))
         dotime(new LinkedHashSet(), "LinkedHashSet");

      else
         dotime(new TreeSet(), "TreeSet");
   }
}
```

Program Output on some CPU

java SetDemo2 HashSet

HashSet 2688 2.688 seconds

With a good hash function, this time will not vary significantly with more elements.

java SetDemo2 LinkedHashSet

LinkedHashSet 4078 4.078 seconds, 1.5X

More expensive due to use of a doubly-linked list to maintain insertion order. However, iteration over a LinkedHashSet will typically be faster than for a HashSet.

java SetDemo2 TreeSet

TreeSet 6469 6.469 seconds, 2.4X

More expensive due to use of red-black trees, which are maintained balanced to keep from being skewed and degrading tree lookup performance.

Example Timing Comparisons for HashSet vs. ArrayList, Vector, & LinkedList

- The input text was the complete text of an issue of *The Guardian* newspaper, from April 1995: 119,604 words; 23,727 unique words.

Collection	Running Time/s
ArrayList	87.4
Vector	92.8
LinkedList	121.5
HashSet	0.2
TreeSet	0.4

Source: http://www.cs.umbc.edu/~jekkin1/os/source/java-lec7-printable.pdf

Week 4 Supplemental Topics

- Java Collections
- **Java data structures**
  - HashSets, etc.
  - **Arrays**
  - Trees & tree processing
- Object-inherited methods

Arrays in Java

- The length of an array is fixed at the time of its creation. An array represents related entities having the same data type in contiguous or adjacent memory locations. The related data items or array elements form a group and are referred to by the same name, e.g. employee[5]; refers to data item 5 in array employee.
- Array declaration example:

  type[] array_name; → String[] designations;
- Array memory allocation example:

  array_name = new type[size] → designations = new String[10];
- Array initialization examples:

```
String[] designations;
designations = new String[2];
designations[0] = "General Manager"; // arrays are always indexed from 0
designations[1] = "Managing Director";
```

or: String[] designations = {"General Manager", "Managing Director"};

Sources: http://www.tutorialhero.com/tutorial-76-java_arrays.php
http://java.sun.com/docs/books/tutorial/java/nutsandbolts/arrays.html

A Simple Array Demo – No Looping

```
class ArrayDemo {
    public static void main(String[] args) {
        int[] anArray; // declares an array of integers

        anArray = new int[10];
        // allocates memory for 10 integers

        anArray[0] = 100; // initialize first element
        anArray[1] = 200; // initialize second
        anArray[2] = 300; // etc.
        anArray[3] = 400;
        anArray[4] = 500;
        anArray[5] = 600;
        anArray[6] = 700;
        anArray[7] = 800;
        anArray[8] = 900;
        anArray[9] = 1000;

        // continued at right …
```

```
        System.out.println("Element at index 0: " + anArray[0]);
        System.out.println("Element at index 1: " + anArray[1]);
        System.out.println("Element at index 2: " + anArray[2]);
        System.out.println("Element at index 3: " + anArray[3]);
        System.out.println("Element at index 4: " + anArray[4]);
        System.out.println("Element at index 5: " + anArray[5]);
        System.out.println("Element at index 6: " + anArray[6]);
        System.out.println("Element at index 7: " + anArray[7]);
        System.out.println("Element at index 8: " + anArray[8]);
        System.out.println("Element at index 9: " + anArray[9]);
    }
}
```

Output:

```
Element at index 0: 100
Element at index 1: 200
Element at index 2: 300
Element at index 3: 400
Element at index 4: 500
Element at index 5: 600
Element at index 6: 700
Element at index 7: 800
Element at index 8: 900
Element at index 9: 1000
```

A Simple Example Array Program

```
import java.io.*;
class Student {
   int regno, total; // registration number, total grade
   int[] grade; // individual grades
   String name; // student name
   public Student(int r, String n, int[] g) {
      regno = r;
      name = n;
      grade = new int[g.length];
      // new used to allocate memory to the array
      for (int i=0; i < g.length; i++) {
         grade[i] = g[i];
         if (grade[i]> 50) total += grade[i];
         else { total = 0; break; }
         // any individual grade <= 50
                  // means total grade = 0
      }
   }
   public void displayStudent() {
         System.out.println("name: " + name);
         System.out.println("regno: " + regno);
         System.out.println("grade total: " + total);
         System.out.println("");
       }
}
```

```
class TestStudent {
   public static void main(String args[]) {
      int[] grade1 = {73, 85, 95}; // declare and initialize
      int[] grade2 = {71, 85, 95};
      Student[] st = new Student[2];
      st[0] = new Student(1, "Ganguly", grade1);
      st[1] = new Student(2, "Sachin", grade2);
      for (int i=0; i < st.length; i++)
         // alternatively: for (Student s : st)     preferred form
         st[i].displayStudent();   // -or- s.displayStudent();
      }
   }
}
```

Output:

```
javac TestStudent.java
java TestStudent

name: Ganguly
regno: 1
grade total: 253

name: Sachin
regno: 2
total grade: 251
```

Multidimensional Arrays

- You can also declare an array of arrays (also known as a *multidimensional* array) by using two or more sets of square brackets, such as String[][] names. Each element, therefore, must be accessed by a corresponding number of index values.
- In the Java programming language, a multidimensional array is simply an array whose components are themselves arrays. This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length, as shown in the following MultiDimArrayDemo program:

```
class MultiDimArrayDemo {
    public static void main(String[] args) {
        String[][] names = {{"Mr. ", "Mrs. ", "Ms. "}, {"Smith", "Jones"}};
    System.out.println(names[0][0] + names[1][0]); // Mr. Smith
    System.out.println(names[0][2] + names[1][1]); // Ms. Jones
     }
}
```

- The output from this program is:

```
Mr. Smith
Ms. Jones
```

Copying Arrays

- The System class has a static arraycopy() method that you can use to efficiently copy data from one array into another:

```
public static void arraycopy(Object src,    // source array to copy from
                             int srcPos,    // starting position in source array
                             Object dest,  // destination array to copy to
                             int destPos,  // starting position in destination
                             int length)    // number of elements to copy
```

- The following demo program copies part of a character array:

```
class ArrayCopyDemo {
   public static void main(String[] args) {
      char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e', 'i', 'n', 'a', 't', 'e', 'd' };
      char[] copyTo = new char[7];
      System.arraycopy(copyFrom, 2, copyTo, 0, 7);
      System.out.println(new String(copyTo)); // converts char array to String
   }
}
```

- The output from the program is:

```
caffein
```

The Arrays Class (java.util.Arrays)

- The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, filling array elements, and printing arrays. These methods are overloaded for all primitive types.

Item Sample of Arrays Methods, with Descriptions

1 **public static int binarySearch(Object[] a, Object key)**

Searches the specified array of Object (or primitive types: byte, int, double, etc) for the specified value using the binary search algorithm. The array must be sorted prior to making this call. Returns the index of the search key, if it is contained in the list; otherwise, -(insertion point + 1).

2 **public static boolean equals(long[] a, long[] a2)**

Returns true if the two specified arrays of longs are equal to one another. Two arrays are considered equal if both arrays contain the same number of elements, and all corresponding pairs of elements in the two arrays are equal. The same method can be used with all other primitive data types (byte, short, int, etc.). There's also **deepEquals()** , which compares multi-dimensional arrays to any arbitrary depth.

3 **public static void fill(int[] a, int val)**

Assigns the specified int value to each element of the specified array of ints. The same method can be used with all other primitive data types (byte, short, long, etc.).

4 **public static void sort(Object[] a)**

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. The same method can also be used with all primitive data types (byte, short, int, etc.).

5 **public static String toString(Object[] a)**

Returns a String representation of the array. The same method can be used by all primitive data types (byte, short, int, etc.). There's also **deepToString()**, which follows elements which are themselves arrays.

varargs and Arrays

- Since Java 5.0, varargs methods accept zero or more arguments of a specified type, converting the arguments into an array:

```
// Simple use of varargs
static int sum(int... args) {
    int sum = 0;
    for (int arg : args)
        sum += arg;
    return sum;
}
```

- If you have a method that requires one or more arguments of a given type, write it this way:

```
// The right way to use varargs to pass one or more arguments
static int min(int firstArg, int... remainingArgs) {
    int min = firstArg;
    for (int arg : remainingArgs)
        if (arg < min)
            min = arg;
    return min;
}
```

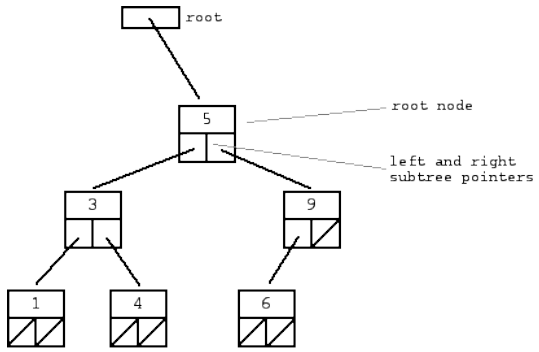
Source: Bloch Item #42

Week 4 Supplemental Topics

- Java Collections
- **Java data structures**
  - HashSets, etc.
  - Arrays
  - **Trees & tree processing**
- Object-inherited methods

Binary Trees and Examples

- A binary tree is made of nodes, where each node contains a "left" pointer, a "right" pointer, and a data element. The "root" pointer points to the topmost node in the tree. The left and right pointers recursively point to smaller "subtrees" on either side. A null pointer represents a binary tree with no elements -- the empty tree. The formal recursive definition is: a **binary tree** is either empty (represented by a null pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**. An example:



- A "binary search tree" (BST) or "ordered binary tree" is a type of binary tree where the nodes are arranged in order: for each node, all elements in its left subtree are less-or-equal to the node (<=), and all the elements in its right subtree are greater than the node (>). The tree shown above is a binary search tree -- the "root" node is a 5, and its left subtree nodes (1, 3, 4) are <= 5, and its right subtree nodes (6, 9) are > 5. Recursively, each of the subtrees must also obey the binary search tree constraint: in the (1, 3, 4) subtree, the 3 is the root, the 1 <= 3 and 4 > 3. Basically, binary trees are fast at insert and lookup, and so are good for “dictionary” problems; they can typically locate a node in log2(N) time.
- The nodes at the bottom edge of the tree have empty subtrees and are called "leaf" nodes (1, 4, 6) while the others are "internal" nodes (3, 5, 9).

Source: http://cslibrary.stanford.edu/110/BinaryTrees.html

A Basic Binary Tree Class

Creation

```
// BinaryTree.java
public class BinaryTree {
 // Root node pointer. Will be null for an empty tree.
 private Node root;

 /*
  --Node--
  The binary tree is built using this nested node class.
  Each node stores one data element, and has a left and right
  sub-tree pointer which may be null.
  The node is a "dumb" nested class -- we just use it for
  storage; it does not have any methods.
 */
 private static class Node {
   Node left;
   Node right;
   int data;

   Node(int newData) {
     left = null;
     right = null;
     data = newData;
   }
 }

 /**
  Creates an empty binary tree -- a null root pointer.
 */
 public void BinaryTree() {
   root = null;
 }
```

Lookup

```
 /**
  Returns true if the given target is in the binary tree.
  Uses a recursive helper.
 */
 public boolean lookup(int data) {
   return(lookup(root, data));
 }

 /**
  Recursive lookup  -- given a node, recurse
  down searching for the given data.
 */
 private boolean lookup(Node node, int data) {
   if (node == null) {
     return(false);
   }

   if (data == node.data) {
     return(true);
   }
   else if (data < node.data) {
     return(lookup(node.left, data));
   }
   else {
     return(lookup(node.right, data));
   }
 }
```

Insertion

```
 /**
  Inserts the given data into the binary tree.
  Uses a recursive helper.
 */
 public void insert(int data) {
   root = insert(root, data);
 }

 /**
  Recursive insert -- given a node pointer, recurse
  down and insert the given data into the tree.
  Returns the new node pointer (the standard way to
  communicate  a changed pointer back to the caller).
 */
 private Node insert(Node node, int data) {
   if (node == null) {
     node = new Node(data);
   }
   else {
     if (data <= node.data) {
       node.left = insert(node.left, data);
     }
     else {
       node.right = insert(node.right, data);
     }
   }

   return(node);
   // in any case, return the new pointer to the caller
 }
```

Example Binary Tree Programs

Build a simple binary tree with 3 notes: 1, 2, 3;
2 is at the top, 1 to the left, and 3 to the right:

```
/**
Build 123 using three pointer variables.
*/
public void build123a() {
 root = new Node(2);
 Node lChild = new Node(1);
 Node rChild = new Node(3);
 root.left = lChild;
 root.right= rChild;
}

/**
Build 123 using only one pointer variable.
*/
public void build123b() {
 root = new Node(2);
 root.left = new Node(1); root.right = new Node(3);
}

/**
Build 123 by calling insert() three times.
Note that the '2' must be inserted first.
*/
public void build123c() {
 root = null; root = insert(root, 2);
 root = insert(root, 1); root = insert(root, 3);
}
```

Count the number of nodes in a binary tree:

```
/**
Returns the number of nodes in the tree.
Uses a recursive helper that recurses
down the tree and counts the nodes.
*/
public int size() {
 return(size(root));
}
private int size(Node node) {
 if (node == null) return(0);
 else {
  return(size(node.left) + 1 + size(node.right));
 }
}
```

Find the maximum root to leaf depth of a binary tree:

```
/**
Returns the max root-to-leaf depth of the tree.
Uses a recursive helper that recurses down to find
the max depth.
*/
public int maxDepth() {
 return(maxDepth(root));
}
private int maxDepth(Node node) {
 if (node==null) {
  return(0);
 }
 else {
  int lDepth = maxDepth(node.left);
  int rDepth = maxDepth(node.right);

  // use the larger + 1
  return(Math.max(lDepth, rDepth) + 1);
 }
}
```

Example Binary Tree Programs – 2

Find the minimum value in a non-empty Binary search tree:

```
/**
Returns the min value in a non-empty binary search tree.
Uses a helper method that iterates to the left to find
the min value.
*/
public int minValue() {
return( minValue(root) );
}
/**
Finds the min value in a non-empty binary search tree.
*/
private int minValue(Node node) {
 Node current = node;
 while (current.left != null) {
   current = current.left;
 }

 return(current.data);
}
```

Print node values in “inorder” order:

```
/**
Prints the node values in the "inorder" order.
Uses a recursive helper to do the traversal.
*/
public void printTree() {
printTree(root);
System.out.println();
}
private void printTree(Node node) {
if (node == null) return;

// left, node itself, right
printTree(node.left);
System.out.print(node.data + "  ");
printTree(node.right);
}
```

Print node values in “postorder” order:

```
/**
Prints the node values in the "postorder" order.
Uses a recursive helper to do the traversal.
*/
public void printPostorder() {
printPostorder(root);
System.out.println();
}
public void printPostorder(Node node) {
 if (node == null) return;

 // first recurse on both subtrees
 printPostorder(node.left);
 printPostorder(node.right);

 // then deal with the node
System.out.print(node.data + "  ");
}
```

Example Binary Tree Programs – 3

Determine if there is a path from the root down to a leaf with a particular sum:

```
/**
Given a tree and a sum, returns true if there is a path
from the root down to a leaf, such that adding up all
the values along the path equals the given sum.
Strategy: subtract the node value from the sum when
recursing down, and check to see if the sum is 0 when
you run out of tree.
*/
public boolean hasPathSum(int sum) {
return( hasPathSum(root, sum) );
}

boolean hasPathSum(Node node, int sum) {
 // return true if we run out of tree and sum==0
 if (node == null) {
   return(sum == 0);
 }
 else {
 // otherwise check both subtrees
   int subSum = sum - node.data;
   return(hasPathSum(node.left, subSum) ||
   hasPathSum(node.right, subSum));
 }
}
```

Print out all root to leaf paths, one per line:

```
/**
Given a binary tree, prints out all of its root-to-leaf paths,
one per line. Uses a recursive helper to do the work.
*/
public void printPaths() {
 int[] path = new int[1000];
 printPaths(root, path, 0);
}
/**
Recursive printPaths helper -- given a node, and an
array containing the path from the root node up to
but not including this node, prints out all the root-leaf
paths.
*/
private void printPaths(Node node, int[] path, int pathLen) {
 if (node == null) return;

 // append this node to the path array
 path[pathLen] = node.data;
 pathLen++;

 // it's a leaf, so print the path that led to here
 if (node.left == null && node.right == null) {
   printArray(path, pathLen);
 }
 else {
 // otherwise try both subtrees
   printPaths(node.left, path, pathLen);
   printPaths(node.right, path, pathLen);
 }
}
```

Utility function for root-to-leaf path printing:

```
/**
Utility that prints ints from an array on one line.
*/
private void printArray(int[] ints, int len) {
 int i;
 for (i=0; i<len; i++) {
  System.out.print(ints[i] + " ");
 }
 System.out.println();
}
```

Example Binary Tree Programs – 4

Change a tree into its mirror image:

```
    4      is changed to ...    4
   / \                         / \
  2   5                       5   2
 / \                         / \
1   3                           1   3
```

Uses a recursive helper that recurses over
the tree, swapping the left/right pointers.

```
*/
public void mirror() {
 mirror(root);
}

private void mirror(Node node) {
 if (node != null) {
  // do the sub-trees
  mirror(node.left);
  mirror(node.right);

  // swap the left/right pointers
  Node temp = node.left;
  node.left = node.right;
  node.right = temp;
 }
}
```

Checks to see if two trees are identical:

```
/*
Compares the receiver to another tree to
see if they are structurally identical.
*/
public boolean sameTree(BinaryTree other) {
return( sameTree(root, other.root) );
}
/**
Recursive helper -- recurses down two trees
in parallel, checking to see if they are identical.
*/
boolean sameTree(Node a, Node b) {
 // 1. both empty -> true
 if (a == null && b == null) return(true);

 // 2. both non-empty -> compare them
 else if (a != null && b != null) {
  return(
    a.data == b.data &&
    sameTree(a.left, b.left) &&
    sameTree(a.right, b.right)
  );
 }
 // 3. one empty, one not -> false
 else return(false);
}
```

Determine the number of structurally unique trees possible for key values 1 … numKeys:

```
/**
For key values 1...numKeys, how many structurally
unique binary search trees are possible that store
those keys?
Strategy: consider that each value could be the root.
Recursively find the size of the left and right subtrees.
*/
public static int countTrees(int numKeys) {
 if (numKeys <=1) {
  return(1);
 }
 else {
  // there will be one value at the root, with whatever
  // remains on the left and right each forming their
  // own subtrees.
  // Iterate through all the values that could be the root...
  int sum = 0;
  int left, right, root;

  for (root=1; root<=numKeys; root++) {
   left = countTrees(root-1);
   right = countTrees(numKeys - root);

   // number of possible trees with this root == left*right
   sum += left*right;
  }

  return(sum);
 }
}
```

Example Binary Tree Programs – 5

Determine if a binary tree is a binary search tree:

```
/**
Tests if a tree meets the conditions to be a
binary search tree (BST).
*/
public boolean isBST() {
 return(isBST(root));
}
/**
Recursive helper -- checks if a tree is a BST
using minValue() and maxValue() (not efficient).
*/
private boolean isBST(Node node) {
 if (node == null) return(true);

 // do the subtrees contain values that do not
 // agree with the node?
 if (node.left!=null && maxValue(node.left) > node.data) return(false);
 if (node.right!=null && minValue(node.right) <= node.data) return(false);

 // check that the subtrees themselves are ok
 return( isBST(node.left) && isBST(node.right) );
}
```

Alternative more efficient solution using recursion:

```
/**
Tests if a tree meets the conditions to be a
binary search tree (BST). Uses the efficient
recursive helper.
*/
public boolean isBST2() {
return( isBST2(root, Integer.MIN_VALUE, Integer.MAX_VALUE) );
}
/**
 Efficient BST helper -- Given a node, and min and max values,
 recurs down the tree to verify that it is a BST, and that all
 its nodes are within the min..max range. Works in O(n) time --
 visits each node only once.
*/
private boolean isBST2(Node node, int min, int max) {
 if (node == null) {
  return(true);
 }
 else {
  // left should be in range  min...node.data
  boolean leftOk = isBST2(node.left, min, node.data);

  // if the left is not ok, bail out
  if (!leftOk) return(false);

  // right should be in range node.data+1..max
  boolean rightOk = isBST2(node.right, node.data+1, max);

  return(rightOk);
 }
}
```

Using TreeSet to Compute a Subset

```
import java.util.*;

   public class SetDemo3 {
      static final int MIN = -5;
      static final int MAX = 5;

      public static void main(String args[]) {
         // create a SortedSet with
         // odd MIN-MAX Integer values
         SortedSet set = new TreeSet();
         for (int i = MIN; i <= MAX; i += 2) {
            set.add(new Integer(i));
         }
         System.out.println("set = " + set);
       // continued at right …
```

```
         // create a subset from 0 (inclusive)
            // up to MAX (exclusive)
            SortedSet subset =
               set.subSet(new Integer(0), new Integer(MAX)); // could be autoboxed
            System.out.println("subset = " + subset);

            // add a new element to the subset
            subset.add(new Integer(0));
            System.out.println("set after add = " + set);
         }
      }
```

Program Output:

```
set = [-5, -3, -1, 1, 3, 5]
subset = [1, 3]        this subset from ≥0 to <5 is backed by the original set
set after add = [-5, -3, -1, 0, 1, 3, 5] element 0 was added to the subset and went into
                the original, since a single set is being maintained
```

Week 4 Supplemental Topics

- Java Collections
- Java data structures
- **Object-inherited methods**
  - **toString, equals, hashCode, clone, compareTo**
  - **finalize, getClass, notify, notifyAll, wait**

Object-Inherited Methods

- Since every Class inherits from Object, there is a complete set of methods that every Class inherits:
  - toString – standard String representation of an object
  - equals – standard test for equality (reference types = same)
  - hashCode – required if object will be stored in a hash table
  - clone – required to implement Cloneable interface
  - compareTo – required for sorted Collections
  - finalize – required for some control over garbage collection
  - getClass – returns the runtime Class of the object
  - notify – used for thread processing
  - notifyAll – used for thread processing
  - wait – used for thread processing (3 overloaded versions)

Object-Inherited Methods

Method Summary	
protected Object	**clone**() Creates and returns a copy of this object.
boolean	**equals**(Object obj) Indicates whether some other object is "equal to" this one.
protected void	**finalize**() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class	**getClass**() Returns the runtime class of an object.
int	**hashCode**() Returns a hash code value for the object.
void	**notify**() Wakes up a single thread that is waiting on this object's monitor.
void	**notifyAll**() Wakes up all threads that are waiting on this object's monitor.
String	**toString**() Returns a string representation of the object.
void	**wait**() Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	**wait**(long timeout) Causes current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	**wait**(long timeout, int nanos) Causes current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

equals() Contract

equals

`public boolean equals(Object obj)`

Indicates whether some other object is "equal to" this one.

The `equals` method implements an equivalence relation:

- It is *reflexive*: for any reference value `x`, `x.equals(x)` should return `true`.
- It is *symmetric*: for any reference values `x` and `y`, `x.equals(y)` should return `true` if and only if `y.equals(x)` returns `true`.
- It is *transitive*: for any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` should return `true`.
- It is *consistent*: for any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the object is modified.
- For any non-null reference value `x`, `x.equals(null)` should return `false`.

The `equals` method for class `Object` implements the most discriminating possible equivalence relation on objects; that is, for any reference values `x` and `y`, this method returns `true` if and only if `x` and `y` refer to the same object (`x==y` has the value `true`).

Parameters:

`obj` - the reference object with which to compare.

Returns:

`true` if this object is the same as the obj argument; `false` otherwise.

See Also:

`Boolean.hashCode()`, `Hashtable`

Object's version of equals is based on object identity

Overriding equals() – Issue 1

```
// Can you spot the bug?
public class Bigram {// a Bigram (pronounced By-gram) is an ordered pair of letters
    private final char first;
    private final char second;
    public Bigram(char first, char second) {
        this.first  = first;
        this.second = second;
    }
    public boolean equals(Bigram b) {
        return b.first == first && b.second == second;
    }
    public int hashCode() {
        return 31 * first + second;
    }

    public static void main(String[] args) {
        Set<Bigram> s = new HashSet<Bigram>();
        for (int i = 0; i < 10; i++)
            for (char ch = 'a'; ch <= 'z'; ch++)
                s.add(new Bigram(ch, ch));
        System.out.println(s.size());// supposed to print 26 (Sets eliminate duplicates)
                                     // (set add() uses equals() to detect duplicates)
                                     // actually prints 260 – what's going on??
    }
}
```

- Object's equals() is not overridden, it's overloaded: its signature is public boolean equals(Object obj)

Source: Bloch Item #36

Use @Override to Catch These Errors

```
@Override public boolean equals(Bigram b) {
    return b.first == first && b.second == second;
}
```

If you insert this annotation and try to recompile the program, the compiler will generate an error message like this:

```
Bigram.java:10: method does not override or implement a method
from a supertype
    @Override public boolean equals(Bigram b) {
    ^
```

You will immediately realize what you did wrong, slap yourself on the forehead, and replace the broken equals implementation with a correct one (Item 8):

```
@Override public boolean equals(Object o) {
    if (!(o instanceof Bigram)) // Note: null is not an instance of any type
        return false;
    Bigram b = (Bigram) o;
    return b.first == first && b.second == second;
}
```

equals() Symmetry – Issue 2

- Symmetry: any two objects must agree on whether they are equal, but cis.equals(s) != s.equals(cis)

```
// Broken - violates symmetry!
public final class CaseInsensitiveString {
    private final String s;

    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = s;
    }

    // Broken - violates symmetry!
    @Override public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                ((CaseInsensitiveString) o).s);
        if (o instanceof String)  // One-way interoperability!
            return s.equalsIgnoreCase((String) o);
        return false;
    }
    ...  // Remainder omitted
}

CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
```

Source: Bloch Item #8

hashCode(): Depends on equals()

hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hashtables such as those provided by `java.util.Hashtable`.

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.

As much as is reasonably practical, the hashCode method defined by class `Object` does return distinct integers for distinct objects. (This is typically implemented by converting the internal address of the object into an integer, but this implementation technique is not required by the Java™ programming language.)

Returns:

a hash code value for this object.

See Also:

`equals(java.lang.Object)`, `Hashtable`

Overriding equals() and hashCode() – Issue 3

- Failing to override hashCode when equals is overridden will violate a provision of the hashCode contract:
  - Whenever it is invoked on the same object more than once during an execution of an application, the hashCode method must consistently return the same integer, provided no information used in equals comparisons on the object is modified. The integer need not remain consistent from one execution of an application to another.
  - **If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. → Depending on Object's equals() won't work.**
  - It is not required that if two objects are unequal according to the equals(Object) method then calling the hashCode method on each of the two objects must product distinct integer results. …

Source: Bloch Item #9

toString() Contract

toString

```
public String toString()
```

Returns a string representation of the object. In general, the `toString` method returns a string that "textually represents" this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method.

The `toString` method for class `Object` returns a string consisting of the name of the class of which the object is an instance, the at-sign character `@', and the unsigned hexadecimal representation of the hash code of the object. In other words, this method returns a string equal to the value of:

```
getClass().getName() + '@' + Integer.toHexString(hashCode())
```

Returns:

a string representation of the object.

- Bloch's advice: always override toString(), and return *all* interesting information from an object

Source: Bloch Item #10

clone() Contract

clone

```
protected Object clone()
                  throws CloneNotSupportedException
```

Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object `x`, the expression:

```
x.clone() != x
```

will be true, and that the expression:

```
x.clone().getClass() == x.getClass()
```

will be `true`, but these are not absolute requirements. While it is typically the case that:

```
x.clone().equals(x)
```

will be `true`, this is not an absolute requirement. Copying an object will typically entail creating a new instance of its class, but it also may require copying of internal data structures as well. No constructors are called.

The method `clone` for class `Object` performs a specific cloning operation. First, if the class of this object does not implement the interface `Cloneable`, then a `CloneNotSupportedException` is thrown. Note that all arrays are considered to implement the interface `Cloneable`. Otherwise, this method creates a new instance of the class of this object and initializes all its fields with exactly the contents of the corresponding fields of this object, as if by assignment; the contents of the fields are not themselves cloned. Thus, this method performs a "shallow copy" of this object, not a "deep copy" operation.

The class `Object` does not itself implement the interface `Cloneable`, so calling the `clone` method on an object whose class is `Object` will result in throwing an exception at run time. The `clone` method is implemented by the class `Object` as a convenient, general utility for subclasses that implement the interface `Cloneable`, possibly also overriding the `clone` method, in which case the overriding definition can refer to this utility definition by the call:

```
super.clone()
```

Returns:

a clone of this instance.

Throws:

`CloneNotSupportedException` - if the object's class does not support the `Cloneable` interface. Subclasses that override the `clone` method can also throw this exception to indicate that an instance cannot be cloned.

`OutOfMemoryError` - if there is not enough memory.

See Also:

`Cloneable`

Source: Bloch Item #11

"Override clone() judiciously"

End of Week 4 Supplemental Topics

- Java Collections
- Java data structures
- Object-inherited methods