Intro to Data Types and Structures

Abstract Data Types
Java Collections Framework
Data Type

- Set of values (the data)

- Set of operations defined on the data.

- Example: the integers (data) and the usual arithmetic operations of addition, subtraction, multiplication, and division.
Abstract Data Type

- A specification of a data type in a formal sense without regard to any particular implementation or programming language.

**Example**: the set of integers \{0, \pm 1, \pm 2, \ldots, \pm \infty\} and the operations \(a + b, a - b, a \times b, a/b\)
A realization has two parts

1. The **interface**, **specification**, or **documentation** of the ADT: What is the purpose of each operation and what is the syntax for using it.

2. The **implementation** of the ADT. How is each operation expressed using the data structures and statements of a programming language.
ADT and OOP Languages

Java correspondence between classes and ADTs

- Java class ↔ ADT
- Javadoc/interface ↔ ADT specification
- instance data fields ↔ ADT data
- instance methods ↔ ADT operations
## Classification of ADT operations

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create operation</strong></td>
<td>Constructor call expression in Java</td>
</tr>
<tr>
<td><strong>Copy operations</strong></td>
<td>(assign, shallow or deep) (depends on the particular ADT)</td>
</tr>
<tr>
<td><strong>Destroy operation</strong></td>
<td>(done by garbage collector in Java)</td>
</tr>
<tr>
<td><strong>Modification operations</strong></td>
<td>(mutable) (change one or more data fields)</td>
</tr>
<tr>
<td><strong>Inquiry operations</strong></td>
<td>(immutable) (inspect one or more data fields)</td>
</tr>
</tbody>
</table>

- **set methods**
- **get methods**

```
Pre- and Post-conditions

Pre-conditions

- Conditions that must be true before an operation is executed in order that the operation completes successfully. They are expressed in terms of the state of the object before the operation is applied to the object. A pre-condition may or may not be needed.

Post-conditions

- Conditions that will be true after an operation completes successfully. They are expressed in terms of the state of the object after the operation has been applied to the object.
Simple ADT examples (1)

**Integer ADT**
- Finite subset of integers: \(-2^{m-1} \leq n \leq 2^{m-1}\)
- Operations: +, -, *, quotient, remainder, ==, <, <=, >, >=, !=, etc.

**floating point ADT**
- Finite subset of rational numbers:
  
  \[
  0 \\
  -3.40 \times 10^{38} \leq x \leq -1.40 \times 10^{-45} \\
  1.40 \times 10^{-45} \leq x \leq 3.40 \times 10^{38}
  \]
- Operations: +, -, *, /, ==, <, <=, >, >=, !=, etc.
Simple ADT examples (2)

Character ADT
- Finite subset of integers (ASCII or Unicode) in the range 0 to 65535 (Unicode)
- Operations: convert to and from lower and upper case, compare two characters using !=, <, <=, ==, >, >=

Boolean ADT
- Two data values: false, true or 0 for false, non-zero for true
- Operations: test for true and test for false
Structured ADT examples (1)

Array ADT

- Data values are arrays \( a_0, a_1, \ldots, a_{n-1} \) where the array elements \( a_k \) belong to some other ADT.

- Operations:
  - get an element given its index
    - Java: \( v = a[k] \);
  - set new value for an element given its index
    - Java: \( a[k] = v; \)

- Arrays can be fixed size or dynamic (varying size)
Structured ADT examples (2)

String ADT

- Like arrays of characters but operations are quite different.

- Immutable strings like Java's `String` class have get operations but no set operations.

- Mutable strings like Java's `StringBuilder` class have both get and set operations.

- Many other operations are usually provided such as substring operations, searching operations, and string comparison operations.
User defined ADT examples (1)

Dynamic array ADT

- Data elements are arrays \([a_0, a_1, \ldots, a_{n-1}]\)
- Mutable ADT with get and set operations.
- Array size is increased as needed by applying a resize operation that doubles the array size when it becomes full.
- In Java the standard arrays cannot be resized. Once they are dimensioned the size cannot be changed.
User defined ADT examples (2)

Bag ADT

- Data elements are bags of elements
- A bag is a container that holds a collection of elements of some type
- There is no defined order on the bag elements as for arrays and duplicate elements are allowed.
- In Mathematics bags are often called multi-sets
- Basic operations are to add or remove an element and a contains operation to see if a specified element is in the bag
/**
 * A simple mutable generic bag ADT.
 * @param <E> type of elements in the bag
 */

public interface Bag<E>
{
    /**
     * Return current number of elements in this bag.
     * @return current number of elements in this bag.
     */
    int size();

    // continued
Bag interface in Java (2)

```java
/**
   * Return true if this bag is empty else false.
   * @return true if this bag is empty else false
   */
boolean isEmpty();

/**
   * Add another element to this bag if possible.
   * @param element the element to add.
   * @return true if add was successful else false.
   */
boolean add(E element);

// continued
```
/**
 * Return true if element was removed.
 * @return true if the element was removed
 * A false return value occurs if element was
 * not in this bag
 */
boolean remove(E element);

/**
 * Check if a given element is in this bag.
 * @param element the element to check
 * @return true if element is in this bag
 */
boolean contains(E element);
}
A fixed size implementation (1)

Once constructed for a given maximum size the size cannot be changed.

```java
public class FixedBag<E> implements Bag<E> {
    // instance data fields will go here

    // Now we design the class constructors
    // They are never specified in the interface

    public FixedBag(int bagSize) {...}
    public FixedBag() {...}
    public FixedBag(FixedBag<E> b) {...}

    // continued
```
A fixed size implementation (2)

// now implement the interface methods

public int size() {...}
public boolean isEmpty() {...}
public boolean add(E element) {...}
public boolean remove(E element) {...}
public boolean contains(E element) {...}

// Include a toString method

public String toString() {...}
Constructing a fixed size bag (1)

Construct a bag containing a maximum of 5 integers and add the integers 1, 2, and 3 using the statements

```java
Bag<Integer> b = new FixedBag<Integer>(5);
b.add(1);
b.add(2);
b.add(3);
System.out.println(b);
```

Note that autoboxing is being used here. The expression `b.add(1)` is really `b.add(new Integer(1))`;

Note that the interface type is used on the left side of the assignment statement. This is "programming to an interface".
Constructing a fixed size bag (2)

Following statements construct a fixed size bag containing the integers 1 to 10 using a for loop

```java
Bag<Integer> bag = new FixedBag<Integer>(10);
for (int k = 1; k <= 10; k++)
    bag.add(k);
```

We can do this using a while loop and the fact that the add method returns false if an element cannot be added.

```java
Bag<Integer> bag = new FixedBag<Integer>(10);
int k = 1;
while (bag.add(k))
    k++;
```
Choosing a data structure

We can use an array data structure to hold the bag elements so the instance data fields are

```java
private E[] data;
private int size;
```

Bag elements are stored in `data[0]`, ..., `data[size-1]` and the remaining elements `data[size]`, ..., `data[data.length-1]` are free.
Constructor implementations

```java
public FixedBag(int bagSize)
{  
    data = (E[]) new Object[bagSize];
    size = 0;
}

public FixedBag()
{  
    this(10);
}

public FixedBag(FixedBag<E> b)
{  
    size = b.size;
    data = (E[]) new Object[b.data.length];
    for (int k = 0; k < size; k++)
        data[k] = b.data[k];
}
```

It is necessary to use the Object type when constructing generic arrays.
Implementation of add method

Since there is no order on the elements in a bag the add method can simply store the element in the next available position data[size].

```java
public boolean add(E element) {
    if (size == data.length) // full bag
    {
        return false
    }
    data[size] = element;
    size = size + 1;
    return true;
}
```
Implementation of remove method

Since there is no order on the elements in a bag there is no need to shift elements down to remove the element (this is $O(n)$). Instead just move data[size-1] to position of element to remove (this is $O(1)$).

```java
public boolean remove(E element) {
    for (int k = 0; k < size; k++) {
        if (data[k].equals(element)) {
            data[k] = data[size - 1];
            size = size - 1;
            return true;
        }
    }
    return false; // not found
}
```
Implementation of contains methods

This method, like remove, uses the equals method to compare elements. Therefore it is necessary that the particular class \( E \) has defined an appropriate \texttt{equals} method.

```java
public boolean contains(E element)
{
    for (int k = 0; k < size; k++)
    {
        if (data[k].equals(element))
            return true;
    }
    return false; // not found
}
```
Implementation of toString method

Here we use a `StringBuilder` to build a string representation of the form `[a,b,c]`

```java
public String toString()
{
    StringBuilder sb = new StringBuilder();
    sb.append("[");
    if (size != 0)
    {
        sb.append(data[0]);
        for (int k = 1; k < size; k++)
        {
            sb.append(",");
            sb.append(data[k]);
        }
    }
    sb.append("]");
    return sb.toString();
}
```
The `size` and `isEmpty` methods

```java
public int size() {
    return size;
}

public boolean isEmpty() {
    return size == 0;
}
```
Dynamic implementation

We introduce a private `resize` method that performs the following operations:

- Create a new array that has twice the capacity of the original array.
- Copy the data in the original full array to this new array.
- Reset the reference to this new array. The original one will be garbage collected.
private void resize()
{
    int newCapacity = 2 * data.length;

    E[] newData = ( E[] ) new Object[newCapacity];

    for (int k = 0; k < data.length; k++)
    {
        newData[k] = data[k];
    }

    data = newData;
}
modified add method

All we need to do is call `resize` if the array is full when `add` is called. Remaining methods are same as `FixedBag<E>`

```java
public boolean add(E element) {
    if (size == data.length) {
        resize();
    }
    data[size] = element;
    size = size + 1;
    return true;
}
```
public class DynamicBag<E> implements Bag<E> {
    // The three constructors are same as for
    // FixedBag<E>, only the class name changes

    // Other methods are the same as for FixedBag<E>
    // except for add

    // private resize() method goes here
}
ALGORITHM MemoryGame()
Make a bag that can hold 5 integers
Generate 5 random integers in the range 1 to 10 
and add them to the bag.
LOOP
Make a copy of the original bag.
Ask user for 5 guesses of numbers in the bag 
and remove the guesses from the bag copy if possible.
IF bag copy is now empty THEN
   EXIT LOOP
END IF
Determine how many guesses are correct.
Tell user how many guesses are correct.
END LOOP
Congratulate user on winning the game.
The array ADT

Array ADT

- Data elements are arrays \([a_0, a_1, \ldots, a_{n-1}]\)
- Unlike the bag there is an order: first element with index 0, second element with index 1 and so on
- add method adds after the end of the array
- get method returns the element with given index
- set method changes the element with given index
- remove, indexOf methods (exercise)
Array interface in Java (1)

```java
package chapter13.arrays;
/**
 * A simple generic array ADT.
 * @param <E> type of elements in the array
 */
public interface Array<E>
{
    /**
     * Return number of elements in this array.
     * @return number of elements in this array.
     */
    int size();

    // continued
```
///
  * Return true if this array is empty else false.
  * @return true if this array is empty else false
  */
  boolean isEmpty();

/**
 * Add another element to end of this array
 * @param element the element to add.
 * @return true if add was successful else false.
 */
  boolean add(E element);

// continued

/**
 * Get element at given index (0,1,...).
 * @param index the index of the element
 * @return the element at the given index
 * @throws ArrayIndexOutOfBoundsException ...
 * 0 <= index < size()
 */

E get(int index);

/**
 * Set a new value for a given array element.
 * @param index the index of the array element
 * @param element the new value of array element
 * @throws ArrayIndexOutOfBoundsException ...
 * 0 <= index < size()
 */

void set(int index, E element);
DynamicArray<E> design

```java
public class DynamicArray<E> implements Array<E> {
    private E[] data;
    private int size;

    public DynamicArray(int initialCapacity) {..}
    public DynamicArray() {...}
    public DynamicArray(DynamicArray<E> a) {...}

    public int size() {...}
    public boolean isEmpty() {...}
    public boolean add(E element) {...}
    public E get(int index) {...}
    public void set(int index, E element) {...}
    public String toString() {...}
}
```
Dynamic array examples (1)

Following example tests that an array is automatically resized when it becomes full and the add method is called.

```java
Array<Integer> a = new DynamicArray<Integer>(3);
a.add(1);
a.add(2);
a.add(3); // should be full now
a.add(4); // expanded to size 6
System.out.println("Array size is " + a.size());
System.out.println("Array is " + a);
```
Following example sums the elements in a dynamic array.

```
Array<Integer> a = new DynamicArray<Integer>(3);
a.add(1); a.add(2); a.add(3); a.add(4);
int sum = 0;
for (int k = 0; k < a.size(); k++)
{
    sum = sum + a.get(k);
}
System.out.println("The sum is " + sum);
```
Following example shows how to swap two elements of an array str of strings and compare with the built-in array type

```java
String temp = str.get(i);
str.set(i, str.get(j));
str.set(j, temp);
```

```java
String temp = str[i];
str[i] = str[j];
str[j] = temp;
```
DynamicArray<E> class (1)

```java
public class DynamicArray<E> implements Array<E>
{
    private E[] data;
    private int size;

    /**
     * Create an array for given initial capacity
     * @param initialCapacity the initial capacity
     */
    public DynamicArray(int initialCapacity)
    {
        data = (E[]) new Object[initialCapacity];
        size = 0;
    }
}
```
DynamicArray<E> class (2)

/**
 * default array for initialCapacity 10
 */
public DynamicArray()
{
    this(10);
}

/**
 * Construct array that is a copy of given array.
 * The copy has same capacity as original.
 * @param a the array to copy
 */
public DynamicArray(DynamicArray<E> a)
{
    size = a.size();
    data = ( E[] ) new Object[a.data.length];
    for (int k = 0; k < size; k++)
        data[k] = a.data[k];
}
DynamicArray<E> class (3)

```java
public int size()
{
    return size;
}

public boolean isEmpty()
{
    return size == 0;
}

public boolean add(E element)
{
    // same as for DynamicBag<E>
}

private void resize() // same as for DynamicBag<E>
public String toString() // same as for FixedBag<E>
```
public E get(int index)
{
    if (0 <= index && index < size)
        return data[index];
    else
        throw new ArrayIndexOutOfBoundsException("...");
}

public void set(int index, E element)
{
    if (0 <= index && index < size)
        data[index] = element;
    else
        throw new ArrayIndexOutOfBoundsException("...");
}
Java Collections Framework (JCF)

JCF interfaces
JCF implementing classes
Many ADT's collect together elements of some data type
Example: array of strings
Example: bag of integers

Collection

- A data type that organizes a group of related objects called the elements of the collection and provides operations on them.
- There are often restrictions on the elements that belong to a specific kind of collection and on the way the elements can be accessed.
JCF interface hierarchy

- `Iterable<E>`
- `Collection<E>`
  - `Set<E>`
  - `List<E>`
  - `SortedSet<E>`
- `Iterator<E>`
- `Map<K,V>`
  - `ListIterator<E>`
  - `SortedMap<K,V>`
Collection summary

Bag ADT
(1) elements have no defined order
(2) duplicate elements are allowed

Set ADT
(1) elements have no defined order
(2) duplicate elements are not allowed

List ADT
(1) elements have a defined order
(2) duplicate elements are allowed

Note: Maps do not extend the Collection<E> interface but their views (keys, values, entries) are collections.
public interface Collection<E> extends Iterable<E> {
    // Query operations

    int size();
    boolean isEmpty();
    boolean contains(Object obj);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);

    // Modification Operations

    boolean add(E element); // optional
    boolean remove(Object obj); // optional
// Bulk operations

boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c); // optional
boolean removeAll(Collection<?> c); // optional
boolean retainAll(Collection<?> c); // optional
void clear(); // optional

// Comparison and hashing

boolean equals(Object obj);
int hashCode();

Any type that extends or implements the type E
The `Iterable<E>` interface is related to the for-each loop introduced in Java 5 and the `Iterator<E>` interface is used to return an iterator for traversing the collection elements.

```java
public interface Iterable<E>
{
    Iterator<E> iterator();
}
```

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

To traverse a collection means to examine or process elements in the collection one at a time.

An iterator is an object from some class that implements the `Iterator<E>` interface.

An iterator can be used to traverse a collection without needing to know the internal implementation details of the collection.
Using an iterator (1)

Assuming that `ACollectionClass<E>` is some class that implements the `Collection<E>` interface the following statements show how to use the iterator for a traversal

```java
// construct a collection object
Collection<E> col = new ACollectionClass<E>(...);
// add some elements to it
col.add(e1); col.add(e2); col.add(e3);  // ...
// get an iterator for the collection
Iterator<E> iter = col.iterator();
// perform a traversal with the iterator
while(iter.hasNext())
{
    E element = iter.next();
    // do something here with element
}
```
The following statements show how an iterator can be used as a filter by removing elements from the collection that satisfy some condition

```java
// construct a collection object
Collection<E> col = new ACollectionClass<E>(...);
// add some elements to the collection
col.add(e1); col.add(e2); col.add(e3); // ...
// Get an iterator for the collection
Iterator<E> iter = col.iterator();
while(iter.hasNext())
{
    E element = iter.next();
    if (removel condition is true )
        iter.remove();
}
```
The following statements show how an iterator can be used as a filter in case the iterator is immutable by creating a second collection and adding elements to it.

```java
// construct two collection objects
Collection<E> col = new ACollectionClass<E>(...);
Collection<E> newCol = new ACollectionClass<E>(...);

// add some elements to one of them
col.add(e1); col.add(e2); col.add(e3); // ...

Iterator<E> iter = col.iterator();
while(iter.hasNext())
{
    E element = iter.next();
    if (removal condition is NOT true)
        newCol.add(element);
}
```
Using the `Iterable<E>` interface

The `Iterable<E>` interface is related to the for-each loop. If a class implements this interface then the for-each loop can be used as an immutable iterator.

```java
for (E element : col)
{
    // do something here with element
}
```

```java
String[] s = new String[3];
s[0] = "one"; s[1] = "two"; s[2] = "three";
for (String str : s)
{
    // do something here with the string str
}
```

also works with the standard array type
There are 4 types of methods:

- Query operations (do not change the collection)
- Modification operations (mutable, change collection)
- Bulk operations (combine with other collections)
- Comparison and hashing operations
Collection<E> method summary (1)

**int size()**

Returns the number of elements in this collection

**boolean isEmpty()**

Returns true if there are no elements in this collection else returns false.

**boolean contains(Object obj)**

Returns true if this collection contains element obj else returns false.
**Iterator<E> iterator()**

Returns an iterator of type `Iterator<E>` for this collection. This is the method that is necessary to implement the `Iterable<E>` interface.

**Object[] toArray()**

Convert the elements in this collection to an array of `Object` type. Example

```java
Object[] s = c.toArray();
```
<T> T[] toArray(T[] a)

This is a parametrized method for type T that returns an array T[] of type T. If the collection does not contain only elements of type T an exception is thrown. Example

String[] s = c.toArray(new String[c.size()]);

converts a collection, c, of strings to the array s of strings

s[0], s[1], ..., s[s.length - 1]
boolean add(E element)

Returns true if this collection was changed (element was added) after calling the method else returns false. This is an optional operation.

boolean remove(Object obj)

Returns true if this collection was changed (obj was found and removed) after calling the method else returns false. This is an optional operation.
boolean containsAll(Collection<?> c)

Returns true if this collection contains all the elements in collection c (wild card type) else returns false.

boolean addAll(Collection<? extends E> c)

Add all the elements of c to this collection. Returns true if this collection was modified else returns false.

The type <? extends E> means any type that extends or implements type E.

This is an optional operation.
boolean removeAll(Collection<?> c)

Returns true if this collection was modified (one or more elements of c were removed from this collection) else returns false.

This is an optional operation.

boolean retainAll(Collection<?> c)

Retains only elements of this collection that are also in c. Returns true if this collection was modified else returns false.
This is an optional operation.
Collection<E> method summary (7)

- **void clear()**
  
  Removes all elements of this collection to give an empty collection.
  
  This is an optional operation.

- **boolean equals(Object obj)**
  
  int hashCode() (7)

  These are methods in the Object class that can be overridden. The equals method tests if two collections have the same elements.
Set<E> interface

```java
public interface Set<E> extends Collection<E> {
    // The Collection<E> methods go here.
    // Set<E> introduces no new methods.
}
```

The Set<E> interface extends Collection<E> but doesn't add any new methods. However the documentation of some methods changes since a set has no order and no duplicates.

For example the `contains` method will return false if the `element` is already in this set and the `add` method will not change the collection if the element is already in this set. Similarly `addAll` will only add to this set the elements of the collection `c` that are not already in this set.
Set theory operations

- **a.containsAll(b)**
  returns true if a is a superset of b

- **a.removeAll(b)**
  replaces a by the difference a – b

- **a.addAll(b)**
  replaces a by the set union of a and b

- **a.retainAll(b)**
  replaces a by the set intersection of a and b

To obtain non-destructive versions it is necessary to first make a copy of a and apply the operation to the copy
Classes that implement `Set<E>`

<table>
<thead>
<tr>
<th>Class</th>
<th>Key Features</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HashSet&lt;E&gt;</strong></td>
<td>- uses hash table data structure</td>
</tr>
<tr>
<td></td>
<td>- fastest implementation</td>
</tr>
<tr>
<td></td>
<td>- order is not preserved</td>
</tr>
<tr>
<td><strong>LinkedHashSet&lt;E&gt;</strong></td>
<td>- uses linked hash table data structure</td>
</tr>
<tr>
<td></td>
<td>- order in which elements are added is maintained</td>
</tr>
<tr>
<td><strong>TreeSet&lt;E&gt;</strong></td>
<td>- uses tree data structure</td>
</tr>
<tr>
<td></td>
<td>- slower</td>
</tr>
<tr>
<td></td>
<td>- If there is a total order defined on the set element type <code>E</code> then this implementation maintains the set in sorted order</td>
</tr>
</tbody>
</table>
public class HashSet<E> extends AbstractSet<E>
   implements Set<E>, Cloneable, Serializable
{
    public HashSet() {...}
    public HashSet(int initialCapacity) {...}
    public HashSet(Collection<? extends E> c) {...}
    public HashSet(int initialCapacity, float loadFactor) {...}

    public Object clone() {...}

    // implementation of Set interface goes here
}
public class LinkedHashSet<E> extends HashSet<E>
  implements Set<E>, Cloneable, Serializable
{
  public LinkedHashSet() {...}
  public LinkedHashSet(int initialCapacity) {...}
  public LinkedHashSet(Collection<? extends E> c)
  {
  ...
  }
  public LinkedHashSet(int initialCapacity,
                         float loadFactor) {...}

  public Object clone() {...}

  // implementation of Set interface goes here
}
public class TreeSet<E> extends AbstractSet<E>
  implements SortedSet<E>, Cloneable, Serializable
{
  
  public TreeSet() { ... }
  public TreeSet(Collection<? extends E> c) { ... }
  public TreeSet(Comparator<? super E> c) { ... }
  public TreeSet(SortedSet<E> s) { ... }

  public Object clone() { ... }

  // implementation of SortedSet<E> goes here
  // SortedSet<E> extends the Set<E> interface
}
Removing duplicates from a collection.
Let \( c \) be a collection of strings and create a new collection called \texttt{noDups} that is \( c \) with duplicates removed.

This can be done with the conversion constructor in the \texttt{HashSet\langle E\rangle} class that implements \texttt{Set\langle E\rangle}.

Any duplicates in \( c \) will not be copied into the set since sets cannot have duplicates.

```java
Set<String> noDups = new HashSet<String>(c);
```
Create a set of 10 integers generated randomly in the range 1 to $n$ where $n > 9$.

This can be done by trying to add elements to the set until the set has size 10.

It is important to have $n > 9$ or the loop will be infinite since there are no sets of size 10 containing only numbers in the range $1 \leq k \leq 9$.

```java
Random random = new Random();
Set<Integer> randomSet = new TreeSet<Integer>();
while (randomSet.size() < 10)
{
    randomSet.add(random.nextInt(n) + 1);
}
```
Using HashSet to compute set union.

```java
Set<String> s1 = new HashSet<String>();
s1.add("one"); s1.add("two"); s1.add("three");
Set<String> s2 = new HashSet<String>();
s2.add("four"); s2.add("five"); s2.add("six");
Set<String> union = new HashSet<String>(s1);
union.addAll(s2);
System.out.println(union);
```

Output using HashSet: [one, two, five, four, three, six]
Output using LinkedHashSet: [one, two, three, four, five, six]
Output using TreeSet: [five, four, one, six, three, two]
Use an iterator to remove all even integers from a set of integers.

```java
Set<Integer> s = new HashSet<Integer>();
s.add(1);s.add(2);s.add(3);s.add(3);s.add(4);
Iterator<Integer> iter = s.iterator();
Set<Integer> evenSet = new HashSet<Integer>();
Set<Integer> oddSet = new HashSet<Integer>();
while (iter.hasNext())
{  int k = iter.next();
    if (k % 2 ==0)
        evenSet.add(k);
    else
        oddSet.add(k);
}
System.out.println(evenSet); ...
```
Write a program that reads a file of words and displays the unique words (duplicates removed).

```java
import java.io.*;
import java.util.*;
public class RemoveDuplicateWords {
    public void doTest() throws FileNotFoundException {
        Scanner input = new Scanner(new File(...));
        Set<String> uniqueSet = new HashSet<String>();
        // The Scanner class implements Iterator<String>
        Iterator<String> iter = input;
        while(iter.hasNext()) {
            uniqueSet.add(iter.next());
        }
        // continued
    }
}
```
Removing duplicate words (2)

```java
input.close();
System.out.println(uniqueSet.size() + " unique words found:");
System.out.println(uniqueSet);
}

public static void main(String[] args)
    throws FileNotFoundException
{
    new RemoveDuplicateWords().doTest();
}
```

all all words words are are duplicated duplicated

4 unique words found: [words, all, duplicated, are]
A list is a collection of elements arranged in some linear order.

Like an array and unlike a set or a bag, a list has a first element, a second element and so on.

The `List<E>` interface extends `Collection<E>` and provides some index based methods to locate elements.

The `ListIterator<E>` interface extends `Iterator<E>` to provide methods that allow traversals in either direction.
public interface List<E> extends Collection<E>
{
    // Collection<E> interface methods can go here

    // Positional Access Operations (index = 0, 1, ...)

    E get(int index);
    E set(int index, E element); // optional
    void add(int index, E element); // optional
    E remove(int index); // optional
    boolean addAll(int index,
                    Collection<? extends E> c); // optional

    // continued
// Search Operations

int indexOf(Object obj);
int lastIndexOf(Object obj);

// List Iterators

ListIterator<E> listIterator();
ListIterator<E> listIterator(int index);

// View

List<E> subList(int fromIndex, int toIndex);
ListIterator<E> interface

public interface ListIterator<E> extends Iterator<E>
{
    // Query Operations
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();

    // Modification Operations
    void remove(); // optional
    void set(E element); // optional
    void add(E element); // optional
}
List<E> method summary (1)

E get(int index)

Return the element in this list at given position index. If index < 0 or index >= size() an index out of bounds exception is thrown.

E set(int index, E element)

Replace element at position index by the given element and return the replaced element. If index < 0 or index >= size() an index out of bounds exception is thrown. This is an optional operation
void add(int index, E element)

Add a new element to this list at position index. The elements originally beginning at position index are moved up to higher indices to accommodate the new element. If index < 0 or index > size() an index out of bounds exception is thrown.

Note that index can be equal to size() here, corresponding to adding an element after the last element.

This is an optional operation.
boolean addAll(int index,
    Collection<? extends E> c)

Add all the elements in the given collection c to this list beginning at the given position index. The elements originally beginning at position index are moved up to higher indices to accommodate the new elements.

The restrictions on index are the same as for the add method.

This is an optional operation.
List<E> method summary (4)

boolean indexOf(Object obj)

Return index of the first occurrence of the given object obj in this list. If obj was not found then -1 is returned.

boolean lastIndexOf(Object obj)

Return index of the last occurrence of the given object obj in this list. If obj was not found then -1 is returned.
List<E> method summary (5)

- **ListIterator<E> listIterator()**
- **ListIterator<E> listIterator(int index)**

Return a `ListIterator<E>` object. For the no-arg version the iterator will start at the beginning of this list. The second version will start at position `index` in this list. The restrictions on `index` are the same as for `get`.

- **List<E> subList(int fromIndex, int toIndex)**

Return a sublist of this list beginning and ending at the given indices. If the indices are not in range an index out of bounds exception is thrown.
ListIterator\(<E>\) indices

\[ e_0 \quad e_1 \quad e_2 \quad \cdots \quad e_n \]

- **next()**
  return element to right of cursor index and advance cursor one place to the right.

- **previous()**
  return element to left of cursor index and advance cursor one place to the left.
The `ArrayList<E>` class implements the `List<E>` interface using a dynamic array data structure.

```java
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, Serializable
{
    public ArrayList() {...}
    public ArrayList(int initialCapacity) {...}
    public ArrayList(Collection<? extends E> c)
    {
    // implementation of List<E> interface
    // methods go here
```
// Extra methods

public Object clone() {...}
public void ensureCapacity(
    int minCapacity) {...}
public void trimToSize() {...}
The `LinkedList<E>` class implements the `List<E>` interface using a linked list data structure.

```java
public class LinkedList<E> extends AbstractSequentialList<E>
    implements List<E>, Queue<E>, RandomAccess, Cloneable, Serializable {

    public LinkedList() {...}
    public LinkedList(Collection<? extends E> c) {...}

    // implementation of List<E> and Queue<E> interface methods go here
```
The above extra methods can be used to write adapter versions of the Stack and Queue interfaces.
Simple list examples (1)

Converting a collection to a list using the conversion constructor in `ArrayList<E>` or `LinkedList<E>` classes. Assume that `c` is any collection of strings.

```java
List<String> list = new ArrayList<String>(c);
```

Append `list2` onto the end of `list1`.

```java
list1.addAll(list2);
```

Append without modifying either `list1` or `list2`.

```java
List<String> list3 =
    new ArrayList<String>(list1);
list3.addAll(list2);
```
Simple list examples (2)

Swapping (exchanging) two list elements

```
String temp = list.get(i);
list.set(i, list.get(j));
list.set(j, temp);
```

Polymorphic method for swapping two list elements

```
public static <E> void swap(List<E> list,
    int i, int j)
{
    E temp = list.get(i);
    list.set(i, list.get(j));
    list.set(j, temp);
}
```
Each book is represented as an object from a Book class.

Each book has data fields for a title, author, price, and number in stock.

The books in the store are organized as a list of type `LinkedList<Book>`

We want to process a list of books and remove the books that are not in stock.
public final class Book implements Comparable<Book>
{
    private String title;
    private String author;
    private double price;
    private int inStock;

    // continued
public Book(String title, String author, double price, int inStock)
{
    this.title = title;
    this.author = author;
    this.price = price;
    this.inStock = inStock;
}

public String getTitle() { return title; }
public String getAuthor() { return author; }
public double getPrice() { return price; }
public int getInStock() { return inStock; }

// continued
public String toString()
{
    return "Book[" + title + "," + author
            + "," + price + "," + inStock + "]";
}

/**
 * Define a natural order using the title.
 * @param b book to compare with this book
 * @return negative, zero, positive results
 */
public int compareTo(Book b)
{
    return title.compareTo(b.title);
}
/**
 * Return true if this book has the same title as obj.
 * @param obj book to compare with this book
 * @return true if this book has same title as obj.
 */

public boolean equal(Object obj)
{
    if (obj == null) ||
        getClass() != obj.getClass())
        return false;
    return title.equals((Book) obj.title);
}

class Book
{
    private String title;

    public Book(String title)
    {
        this.title = title;
    }

    public String getTitle()
    {
        return title;
    }

    public void setTitle(String title)
    {
        this.title = title;
    }

    @Override
    public boolean equals(Object obj)
    {
        if (obj == this) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Book book = (Book) obj;
        return Objects.equals(title, book.getTitle());
    }

    @Override
    public int hashCode()
    {
        return Objects.hash(title);
    }

    @Override
    public String toString()
    {
        return "Book [title=" + title + "]";
    }
}
Method to split a book list: remove books not in stock and create a new reOrderList containing books not in stock.

```java
public static List<Book> reOrderBooks(
    List<Book> list)
{
    List<Book> reOrderList = new LinkedList<Book>();
    Iterator<Book> iter = list.iterator();
    while (iter.hasNext())
    {
        Book b = iter.next();
        if (b.getInstock() == 0)
        {
            reOrderList.add(b);
            iter.remove(); // remove from list
        }
    }
    return reOrderList;
}
```
import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;

public class BookList
{
    /**
     * Modify original list so it contains only books
     * in stock and create a new list that contains books
     * which are out of stock.
     */
public void processBookList()
{
    List<Book> list = new LinkedList<Book>();
    list.add(new Book("Dead Souls", "Ian Rankin", 25.95, 10));
    list.add(new Book("Stranger House", "Reginald Hill", 29.50, 0));
    list.add(new Book("Not Safe After Dark", "Peter Robinson", 32.99, 10));
    list.add(new Book("Original Sin", "P. D. James", 39.95, 0));
    list.add(new Book("Fleshmarket Close", "Ian Rankin", 25.00, 0));
    List<Book> reOrderList = reOrderBooks(list);
    System.out.println("Re-order list:");
    displayList(reOrderList);
    System.out.println("List in stock:");
    displayList(list);
}
/**
 * Create lists of books in stock and reorder list.
 * @param list the book list
 * @return the list of books to be ordered.
 * The original list now contains only books in stock.
 */

public static List<Book> reOrderBooks(List<Book> list)
{
    List<Book> reOrderList = new LinkedList<Book>();
    Iterator<Book> iter = list.iterator();
    while (iter.hasNext())
    {
        Book b = iter.next();
        if (b.getInStock() == 0)
        {
            reOrderList.add(b);
            iter.remove();
        }
    }
    return reOrderList;
}
public static <E> void displayList(List<E> list) {
    for (E element : list) {
        System.out.println(element);
    }
}

public static void main(String[] args) {
    BookList books = new BookList();
    books.processBookList();
}
Suppose \([e_0, e_1, \ldots, e_n]\) is a list that is sorted in some order. We want to insert an element \(e\) into this list in the correct position so that the list remains sorted.

**Algorithm**

- Iterate through the list and compare \(e\) with \(e_k\).
- The iteration continues until we arrive at an element \(e_k\) such that \(e \leq e_k\). Then the proper place for \(e\) is just before \(e_k\).

- Special cases: (1) List is empty (create 1-element list \([e]\)), (2) We never find an \(e_k\) such that \(e \leq e_k\) so \(e\) must be added at the end to get the list \([e_0, e_1, \ldots, e_n, e]\)
public static <E extends Comparable<E>>
void insertInSortedList(List<E> list, E newElement)
{
    ListIterator<E> iter = list.listIterator();
    if (!iter.hasNext()) // empty list case
    {
        iter.add(newElement);
        return;
    }

    while (iter.hasNext())
    {
        E element = iter.next();
        if (newElement.compareTo(element) <= 0)
        {
            iter.previous(); // backup
            iter.add(newElement);
            return;
        }
    }

    iter.add(newElement); // add after end of list
}
import java.util.*;
public class SortedListExample
{
    public void doTest()
    {
        // Try it on a list of strings

        List<String> strList = new ArrayList<String>();
        strList.add("Fred"); strList.add("Jane");
        strList.add("Mike");
        System.out.println(strList);
        insertInSortedList(strList, "Gord");
        System.out.println(strList);
        insertInSortedList(strList,"Carol");
        System.out.println(strList);
        insertInSortedList(strList,"Bob");
        System.out.println(strList);
        insertInSortedList(strList,"Susan");
        System.out.println(strList);
    }
}
SortedListExample (2)

```java
// Try it on a list of books
List<Book> list = new ArrayList<Book>();
insertInSortedList(list,
    new Book("Dead Souls",
        "Ian Rankin", 25.95 ,10));
insertInSortedList(list,
    new Book("Stranger House",
        "Reginald Hill", 29.50 ,0));
insertInSortedList(list,
    new Book("Not Safe After Dark",
        "Peter Robinson", 32.99 ,10));
insertInSortedList(list, new Book("Original Sin",
        "P. D. James", 39.95 ,0));
insertInSortedList(list,
    new Book("Fleshmarket Close",
        "Ian Rankin", 25.00 ,0));
displayList(list);
```
public static <E extends Comparable<E>>
void insertInSortedList(List<E> list, E newElement)
{
    ListIterator<E> iter = list.listIterator();

    if (!iter.hasNext()) // make a 1-element list
    {
        iter.add(newElement);
        return;
    }
}
// Note: when we know where to insert
// the new element we have gone one
// position too far so previous is needed.

while (iter.hasNext())
{
    E element = iter.next();
    if (newElement.compareTo(element) <= 0)
    {
        iter.previous(); // backup
        iter.add(newElement);
        return;
    }
    iter.add(newElement); // add after end of list
}
public static <E> void displayList(List<E> list) {
    for (E element : list)
        System.out.println(element);
}

public static void main(String[] args) {
    SortedListExample example = new SortedListExample();
    example.doTest();
}

A map is a function $f$ that associates elements of one set $K$ called the domain of the map to elements of another set $V$ called the range of the map. Notation is $f: K \rightarrow V$.

Each element of the domain is often called a **key** and the corresponding element of the range is often called the **value**.

The set of key-value pairs $(k, v)$ is denoted by

$$f = \{(k_1, v_1), (k_2, v_2), \ldots, (k_n, v_n)\}$$

The pair $(k, v)$ can also be denoted by $v_k$.

The keys form a set $\{k_1, k_2, \ldots, k_n\}$.

The values form a collection (bag) $v_1, v_2, \ldots, v_n$. 
Consider a set of names of people and their ages. The names are the keys and the values are the ages. An example map is

\[
age = \{(\text{Jane}, 12), (\text{Fred}, 10), (\text{Mary}, 15), (\text{Bob}, 10)\}
\]

<table>
<thead>
<tr>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jane</td>
<td>12</td>
</tr>
<tr>
<td>Fred</td>
<td>10</td>
</tr>
<tr>
<td>Mary</td>
<td>15</td>
</tr>
<tr>
<td>Bob</td>
<td>10</td>
</tr>
</tbody>
</table>

Any map can be represented as a two column table. The first column gives the keys and the second column gives the values.

In OOP the values can be from any ADT even another map but the keys should be objects from an immutable class.
Basic operations on a map

- **add**
  Add a new key-value pair to the map

- **delete**
  Remove a key-value pair given its key

- **replace**
  Replace value in a key-value pair given its key

- **search**
  Search for (look up) the value associated with a given key
Implementation of a map

A naïve approach would be to use an array data structure to store the key-value pairs and, given a key, use a linear search to locate the ordered pair containing this key and hence the value. This is an O(n) algorithm.

Maps are most efficiently implemented using a hash table data structure. A hash code is associated with each key. Using this code the key-value pair can easily be located and under most conditions searching is an O(1) algorithm.

A full discussion of hash tables is beyond the scope of these notes.

We will consider only a very simple example.
A simple hash table example (1)

Suppose the keys and values are both integers.
Consider an array with indices 0 to 10 that can hold the key-value pairs.
Assume that each array location can hold a key-value pair using the notation $v_k$ to indicate the value associated with the given key $k$.
For each key we need a function to transform the key into an array index which can then be used to obtain the value associated with the key.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>
A simple hash table example (2)

Suppose keys are in the range $0 \leq k \leq 1000$
We cannot use the key as the array index since the array has only room for 11 key-value pairs.
Therefore we first map the keys to the range 0 to 10 using the hash code $h(k) = k$ and $i = h(k) \mod 11$.
Try this with the keys 15, 558, 32, 132, 102, 5
Taking these keys $\text{mod } 11$ gives the array indices 4, 8, 10, 0, 3, 5 which are all different.
This gives the following array

<table>
<thead>
<tr>
<th>$v_{132}$</th>
<th>$v_{102}$</th>
<th>$v_{15}$</th>
<th>$v_{5}$</th>
<th>$v_{558}$</th>
<th>$v_{32}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Now try to insert a pair with key 257.

257 mod 11 = 4 and location 4 is already occupied.

This is called a **collision**.

We need what is called a **collision resolution policy**.

The simplest one is to find the next highest empty location, wrapping around if necessary, and store the pair there.

In our example index 6 is the next highest empty location so we store the pair there.

If there is no free location the array must be expanded.
The JCF has a `Map<K,V>` interface that defines the basic operations on maps using two parametrized types, `K` for the keys and `V` for the values.

```java
public interface Map<K,V>
{
    // Query operations

    int size();
    boolean isEmpty();
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    V get(Object key);

    // continued
}
```
The `Map<K,V>` interface (2)

// Modification Operations

V put(K key, V value); // optional
V remove(Object key); // optional

// Bulk Operations

void putAll(
    Map<? extends K, ? extends V> t); //optional
void clear(); // optional

// continued
The `Map<K,V>` interface (3)

// Inner interface for map entries (key-value pairs)

interface Entry(K,V>
{
    K getKey();
    V getValue();
    V setValue(V value); // optional
    boolean equals(Object obj);
    int hashCode();
}

// Views (Maps are not collections but views are)

Set<K> keySet();
Collection<V> values();
Set<Map.Entry<K,V>> entrySet();
// Comparison and hashing

boolean equals(Object obj);
int hashCode();
}
Map<K,V> method summary (1)

- int size();
  Return number of pairs (entries) currently in this map

- boolean isEmpty();
  Return true if this map is empty (contains no entries)

- boolean containsKey(Object key);
  Return true if an entry with the given key is in this map

- boolean containsValue(Object value);
  Return true if an entry with the given value is in this map
Map<K,V> method summary (2)

V get(Object key);

Return the value associated with the given key. This is the "look up" operation. A return value of null either indicates that there is no entry with this key or there is an entry but its value is null.

V put(K key, V value);

Add a new pair (entry) with the given key and value. If the entry was already in this map then the old value is replaced by value and the old value is returned. Otherwise a new entry is added to this map and null is returned. This is an optional operation.
Map<K, V> method summary (3)

V remove(Object key);

If the entry with the given key is in this map then it is removed and its value is returned, else null is returned.

This is an optional operation.

V putAll(Map<? extends K, ? extends V> t);

All the entries in map t are put into this map. The types of the map t can be K and V or any types that extend or implement K and V.

This is an optional operation.
Map<K,V> method summary (4)

void clear();
Remove all entries from this map (gives empty map).
This is an optional operation.

interface Entry<K,V>

K getKey();
return key of this entry

V getValue();
return value of this entry

V setValue(V value);
Set value of this entry to value (optional)

boolean equals(Object obj);
Return true if obj equal to this entry

int hashCode();
Return hash code of this entry
Map<K,V> method summary (5)

- **Set<K> keySet();**
  
  Return keys in this map as a set.

- **Collection<V> valueSet();**
  
  Return values in this map as a collection.

- **Set<Map.Entry<K,V>> entrySet();**
  
  Return entries (key-value pairs) of this map as a set of elements of type Map.Entry<K,V>.
Map<K,V> method summary (6)

- **boolean equals(Object obj);**
  Return true if `obj` is a map equal to this map.

- **int hashCode();**
  Return values in **this** map as a collection.
# Classes that implement Map\(<K,V>\>

The JCF has several implementations of the `Map` interface:

**HashMap\(<K,V>\)**
- Uses hash table data structure
- Fastest implementation
- Order is not preserved

**LinkedHashMap\(<K,V>\)**
- Uses linked hash table data structure
- Order in which elements are added is maintained

**TreeMap\(<K,V>\)**
- Uses tree data structure
- Slower
- If there is a total order defined on the keys then this implementation maintains keys in sorted order
HashMap<K,V> class

```java
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {

    public HashMap() {...}
    public HashMap(int initialCapacity) {...}
    public HashMap(Map<? extends K,? extends V> m) {...}
    public HashMap(int initialCapacity,
                    float loadFactor) {...}

    public Object clone() {...}

    // Implementations of Map interface go here
}
```

Conversion constructor
public class LinkedHashMap<K,V> extends HashMap<K,V>
    implements Map<K,V>, Cloneable, Serializable
{
    public LinkedHashMap() {...}
    public LinkedHashMap(int initialCapacity) {...}
    public LinkedHashMap(Map<? extends K,? extends V> m)
    {
    }
    public LinkedHashMap(int initialCapacity,
                          float loadFactor) {...}
    public LinkedHashMap(int initialCapacity,
                          float loadFactor, boolean accessOrder) {...}
    public Object clone() {...}

    // Implementations of Map interface go here
    // Other methods go here
}
public class TreeMap<K,V> extends AbstractMap<K,V>
    implements SortedMap<K,V>, Cloneable, Serializable
{
    public TreeMap() {...}
    public TreeMap(Comparator<? super K> c) {...}
    public TreeMap(Map<? extends K,? extends V> m) {...}
    public TreeMap(SortedMap<? extends K, extends V> m) {...}

    public Object clone() {...}

    // Implementations of SortedMap interface go here
    // SortedMap extends the Map interface
}
Simple map examples (1)

- Construct a name-age map using Strings for the names and integers for the age.
- Autoboxing and unboxing is used.
- The no-arg constructor has a default size of 16 entries.

Map<String, Integer> age =
    new HashMap<String, Integer>();
age.put("Jane", 12);
age.put("Fred", 10);
age.put("Mary", 15);
age.put("Bob", 10);
System.out.println(age);
Simple map examples (2)

Output using **HashMap** is (you may get different order)

```
[Bob=10, Jane=12, Fred=10, Mary=15]
```

Output using **LinkedHashMap** (order entered)

```
[Jane=12, Fred=10, Mary=15, Bob=10]
```

Output using **TreeMap** (alphabetical order of keys)

```
[Bob=12, Fred=10, Jane=12, Mary=15]
```
Simple map examples (3)

Looking up the age of a given person

```java
String name = "Mary";
int a = age.get(name);
System.out.println("Age of " + name + " is " + a);
```

Using `get` if name is not in the map

```java
String name = "Gord";
Integer a = age.get(name);
System.out.println("Age of " + name + " is " + a);
```

We needed to use `Integer` here to obtain a null answer. Otherwise there is an attempt to unbox null which causes a `NullPointerException`. This only happens with primitive types.
Simple map examples (4)

Checking if a map contains a key

```java
String name = "Jill";
if (age.containsKey(name))
    System.out.println(name + " was found");
else
    System.out.println(name + " was not found");
```
Update a value given its key

```java
String name = "Fred";
age.put(name, 15); // update age from 10 to 15
System.out.println("New age of " + name + " is " + age.getName());
```

Add one year to Fred's age

```java
String name = "Fred";
int currentAge = age.get(name);
age.put(name, currentAge + 1);
System.out.println("New age of " + name + " is " + age.getName());
```
Delete an entry given its key

```java
String name = "Fred";
if (age.containsKey(name))
    age.remove(name);
System.out.println(age);
```

Output is

```
[Bob=10, Jane=12, Mary=15]
```
Iterating over the keys

```
Set<String> keys = age.keySet();
Iterator<String> iter = keys.iterator();
while (iter.hasNext())
{
    String name = iter.next();
    int a = age.get(name);
    System.out.println(name + " -> " + a);
}
```

Output is

Bob -> 10
Jane -> 12
Fred -> 10
Mary -> 15
Iterating over the keys using a for-each loop

```java
for (String name : age.keySet())
{
    System.out.println(name + " -> "
    + age.get(name));
}
```

Output is

Bob -> 10
Jane -> 12
Fred -> 10
Mary -> 15
Compute average age

```java
Set<String> keys = age.keySet();
double sum = 0.0;
for (String name : keys)
{
    sum += age.get(name);
}
System.out.println("Average age is "+ sum / keys.size());
```
Using the `Map.Entry` interface to display map

```java
Set<Map.Entry<String,Integer>> entries = age.entrySet();
Iterator<Map.Entry<String,Integer>> iter = entries.iterator();

while (iter.hasNext())
{
    Map.Entry<String,Integer> entry = iter.next();
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```
Simple map examples (11)

Using the `Map.Entry` interface and the for-each loop

```java
for (Map.Entry<String, Integer> entry : age.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```
Simple map examples (12)

Using Map.Entry to add 1 to all the ages

```java
Set<Map.Entry<String,Integer>> entries = age.entrySet();
Iterator<Map.Entry<String,Integer>> iter = entries.iterator();

while (iter.hasNext())
{
    Map.Entry<String,Integer> entry = iter.next();
    entry.setValue(entry.getValue() + 1);
}
System.out.println(entries);
```
Consider a file called hours.txt whose lines contain a person's name and the number of hours they have worked.

Each person can appear more than once. The problem is to display the total hours worked by each person.

Example input:

Fred:10  
Gord:20  
Fred:30  
Mary:15  
Gord:13  
Mary:4   
Mary:6   

Example output:

Fred -> 40.0  
Mary -> 25.0  
Gord -> 33.0  

Use split(":") from String class
# Hours worked example (2)

## Algorithm

- Read the file into a map of name-hours pairs.
- Each time a line of the file is read check if the name is already in the map.
- If the name is not in the map then add a new entry.
- If the name is in the map then update the hours worked by adding the value read from the file.
Before reading the file create the following map.

```java
Map<String, Double> map = new HashMap<String, Double>();
```

Now if name and hours are the values read from a line in the file the map can be updated using

```java
if (map.containsKey(name)) { // update hours worked
    double currentHours = map.get(name);
    map.put(name, currentHours + hours);
}
else
    map.put(name, hours); // new entry
```
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

public class HoursWorked {
    private static final File IN_FILE = new File("files/hours.txt");

    // continued
public void processFile() throws IOException {
    Map<String, Double> map =
        new HashMap<String, Double>();
    BufferedReader in =
        new BufferedReader(
            new FileReader(IN_FILE));
    String line;

    // continued
while ( (line = in.readLine()) != null) {
    String[] s = line.split(":");
    String name = s[0].trim();
    double hours =
        Double.parseDouble(s[1].trim());

    // Echo for checking

    System.out.println(name + "":"" + hours);
HoursWorked program (4)

// put entries in map and update hours
if (map.containsKey(name))
{
    // update hours worked
    double currentHours = map.get(name);
    map.put(name, currentHours + hours);
}
else // new entry
{
    map.put(name, hours);
}

in.close();
// Display the map, one entry per line

System.out.println("Map is");
for (String name : map.keySet()){
    double hours = map.get(name);
    System.out.println(name + " -> " + hours);
}

public static void main(String[] args)
    throws IOException
{
    new HoursWorked().processFile();
}

Start with a text file called cities.txt. Here is an example:

- Toronto:Canada
- Chicago:USA
- Frankfort:Germany
- Sudbury:Canada
- Venice:Italy
- Acapulco:Mexico
- Berlin:Germany
- Barcelona:Spain
- Los Angeles:USA
- Vancouver:Canada
- Rome:Italy
- Miami:USA
- London:UK
- Mexico City:Mexico
- Madrid:Spain
- Florence:Italy
<table>
<thead>
<tr>
<th>Country</th>
<th>Cities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
<td>Sudbury, Toronto, Vancouver</td>
</tr>
<tr>
<td>Germany</td>
<td>Berlin, Frankfort</td>
</tr>
<tr>
<td>Italy</td>
<td>Florence, Rome, Venice</td>
</tr>
<tr>
<td>Mexico</td>
<td>Acapulco, Mexico City</td>
</tr>
<tr>
<td>Spain</td>
<td>Barcelona, Madrid</td>
</tr>
<tr>
<td>UK</td>
<td>London</td>
</tr>
<tr>
<td>USA</td>
<td>Chicago, Los Angeles, Miami</td>
</tr>
</tbody>
</table>
The output is a map from `String` to `List<String>`

```java
Map<String, List<String>> map =
    new TreeMap<String, List<String>>(){};
```

Write a class called `Cities` to solve this problem. Use an `ArrayList<String>` for each list.

The `Collections` class has static methods for searching and sorting lists. Use one to sort the lists of cities.
The values of a map can be anything even another map.

Problem: develop a map that can record the favorite song, food, golfer, etc, associated with each person.

The key-value pairs of the primary map are names and references to the individual favorite maps.

Each favorite map has key-value pairs that are the category names, such as food, song, and golfer, and the values are the preferences.
Using the sets of ordered pairs notation here is an example

\[\text{favorites} = \{(\text{Bob}, f_1), (\text{Fred}, f_2), (\text{Gord}, f_3)\}\]
\[f_1 = \{(\text{food, salad}), (\text{golfer, Vijay Singh}), (\text{song, White Wedding})\}\]
\[f_2 = \{(\text{food, steak}), (\text{golfer, Tiger Woods}), (\text{song, Satisfaction})\}\]
\[f_3 = \{(\text{food, spaghetti}), (\text{golfer, Phil Mickelson}), (\text{song, Money})\}\]
Favorites map of maps example (3)

- **Bob**
  - **Favorite**: salad
  - **Favorite**: Vijay Singh
  - **Favorite**: White Wedding
- **Fred**
  - **Favorite**: steak
  - **Favorite**: Tiger Woods
  - **Favorite**: Satisfaction
- **Gord**
  - **Favorite**: spaghetti
  - **Favorite**: Phil Mickelson
  - **Favorite**: Money

**Diagram**:
- **Name**
  - Bob
  - Fred
  - Gord

- **Favorite**
  - salad
  - Vijay Singh
  - White Wedding
  - steak
  - Tiger Woods
  - Satisfaction
  - spaghetti
  - Phil Mickelson
  - Money
Now construct the favorite maps (categories, preferences)

Map<String, String> f1 =
    new HashMap<String, String>();
f1.put("golfer", "Vijay Singh");
f1.put("song", "White Wedding");
f1.put("food", "salad");
Favorites map of maps example (5)

Map<String,String> f2 =
    new HashMap<String,String>();
f2.put("golfer", "Tiger Woods");
f2.put("song", "Satisfaction");
f2.put("food", "steak");

Map<String,String> f3 =
    new HashMap<String,String>();
f3.put("golfer", "Phil Mickelson");
f3.put("song", "Money");
f3.put("food", "spaghetti");
Now construct the favorites map. It's a map from strings to maps of strings to strings.

```java
Map<String,Map<String,String>> favorites = new HashMap<String,Map<String,String>>();
```

Now associate the three favorite maps with the three people:

```java
favorites.put("Bob", f1);
favorites.put("Fred", f2);
favorites.put("Gord", f3);
```
Display Fred's favorite map

```java
System.out.println(favorites.get("Fred"));
```

Display Bob's favorite golfer

```java
String golfer = favorites.get("Fred").get("golfer");
System.out.println(golfer);
```

Change Fred's favorite food to chicken

```java
favorites.get("Fred").put("food", "chicken");
```
For-each loop for favorites map.

```java
for (String name : favorites.keySet())
{
    System.out.println(name);
    System.out.println(favorites.get(name));
}
```

Output is

```
Bob
{golfer=Vijay Singh, food=salad, song=White Wedding}
Fred
{golfer=Tiger Woods, food=steak, song=Satisfaction}
Gord
{golfer=Phil MIckelson, food=spaghetti, song=Money}
```
Nested for-each loops.

```java
for (String name : favorites.keySet())
{
    System.out.println("favorites for " + name + ":");
    Map<String,String> favorite = favorites.get(name);
    for (String category : favorite.keySet())
    {
        String preference = favorite.get(category);
        System.out.println("   " + category + ": "
                         + preference);
    }
}
```

Output is

```
favorites for Bob:
  food: salad
  golfer: Vijay Singh
  song: White Wedding
  ...
```
Recursion examples using maps

- Consider a sequence \([s_m, s_{m+1}, s_{m+2}, \ldots, s_n, s_{n+1}, \ldots]\) with starting index \(m\) which is often taken to be 0.

- Many sequences are defined by first-order recurrence relations of the form \(s_n = f(s_{n-1})\)

- It is also common to define sequences by second-order recurrence relations of the form \(s_n = f(s_{n-1}, s_{n-2})\)

- A simple first-order relation is \(s_n = n s_{n-1}\) with \(s_0 = 1\) and the exact solution is the factorial function \(s_n = n!\)

- Many recurrence relations do not have exact solutions so it is necessary to compute terms using the recurrence relation
Fibonacci sequence (1)

\[ F_n = F_{n-1} + F_{n-2}, \quad F_0 = F_1 = 1 \]

```java
public long fib(int n)
{
    if (n == 0 || n == 1)
        return 1L;
    else
        return fib(n-1) + fib(n-2);
}
```

This is a simple recursive algorithm but it is very inefficient because values of the sequence are recalculated many times. For example in the calculation of \( \text{fib}(30) \) the term \( \text{fib}(10) \) is calculated 10,946 times.
We can avoid this recalculation using a technique called memoization. When we encounter $\text{fib}(k)$ for the first time we calculate it using the recurrence relation and store it in a table. When we encounter $\text{fib}(k)$ again we simply look it up in the table. The table can be created using a map of type `Map<Integer,long>` To initialize the map we store the values $f(0) = 1$ and $f(1) = 1$ in it.
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class Fibonacci {
    Map<Integer,Long> m;

    public void calculate() {
        // Create map and initialize it
        // for fib(0)= 1 and fib(1)= 1

        m = new HashMap<Integer,Long>();
        m.put(0,1L);
        m.put(1,1L);
// Get the value of n

Scanner input = new Scanner(System.in);
System.out.println("Enter n");
int n = input.nextInt();

// Time the calculation of fib(n)

long startTime = System.nanoTime();
System.out.println(fib(n));
long time = System.nanoTime() - startTime;

double seconds = (double) time * 1e-9;
System.out.println(seconds);
Fibonacci sequence (5)

public long fib(int n)
{
    if (! m.containsKey(n))
        m.put(n, fib(n-1) + fib(n-2));
    return m.get(n);
}

public static void main(String[] args)
{
    new Fibonacci().calculate();
}

# Fibonacci sequence (6)

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Without memoization</td>
<td>53.8 seconds</td>
</tr>
<tr>
<td>With memoization</td>
<td>$3.43 \times 10^{-4}$ seconds</td>
</tr>
</tbody>
</table>
The Q-sequence (1)

\[ Q(n) = Q(n - Q(n-1)) + Q(n - Q(n-2)), \quad Q(1) = Q(2) = 1 \]

```java
public int q(int n) {
    if (n <= 2) {
        return 1;
    } else {
        return fib(n-1) + fib(n-2);
    }
}
```
import java.util.Map;
import java.util.HashMap;
import java.util.Scanner;

public class QSequence
{
    Map<Integer,Integer> m;

    public void calculate()
    {
        // Create map and initialize it
        // for q(1) = 1 and q(2) = 1
        m = new HashMap<Integer,Integer>();
        m.put(1,1);
        m.put(2,1);
// Get value of n

Scanner input = new Scanner(System.in);
System.out.println("Enter n");
int n = input.nextInt();

// Calculate q(n) and time it

long startTime = System.nanoTime();
System.out.println(q(n));
long time = System.nanoTime() - startTime;
double seconds = (double) time * 1e-9;
System.out.println(seconds);
The Q-sequence (4)

// Calculate q(n) recursively using // memoization

public int q(int n)
{
    if (! m.containsKey(n))
        m.put(n, q(n - q(n-1)) + q(n - q(n-2)));
    return m.get(n);
}

public static void main(String[] args)
{
    new QSequence().calculate();
}

The Q-sequence (5)

Results for q(45)

Without memoization 75.8 seconds

With memoization $4.35 \times 10^{-4}$ seconds
It is like the Math class: a set of useful static methods such as sorting and searching that operate on sets, lists, and maps in the JCF.

There are 50 methods in this class and we summarize here only a few.

```java
static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)
```

Search list of type T for a given key. The list must be in the order specified by the Comparable interface implemented by the list. Returns the zero-based index where key was found or (-index – 1) where index is the location where key could be inserted.
static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)

Like the previous version of `binarySearch` except using the specified implementation of `Comparator` to define the order of the list elements (`list` does not need to implement `Comparable` in this version of `binarySearch`).
static <T extends Comparable<? super T>>
void sort(List<T> list)

Sort the given list into increasing order using the implementation of the Comparable<E> interface provided by the list elements.

static <T> void sort(List<T> list, Comparator<? super T> c)

Like the above version of sort except using the specified implementation of Comparator to define the order (list does not need to implement Comparable in this version of sort).
Arrays utility class

This is a class of static methods like Collections except these methods operate on arrays instead of collections
Our **Book** class implements **Comparable<Book>** which defines the natural order in increasing alphabetical order by title.

We can sort a book list simply by using

```
Collections.sort(list);
```

If we want to use an order other than the natural one it is necessary to write a class that implements the **Comparator<Book>** interface.
The following class defines an order on books in increasing alphabetic order by author.

```java
public class BookComparator implements Comparator<Book> {
    public int compare(Book b1, Book b2) {
        return b1.getAuthor().compareTo(b2.getAuthor());
    }
}
```

We can sort a book list in this order using

```java
Collections.sort(list, new BookComparator());
```
Template 1

This is left justified text

```java
public class Test {
    System.out.println("Hello");
}
```
Template 2

left justified text

Title

description