CSE 1325: Object-Oriented Programming

Lecture 15

Java Class Library (JCL)

Collections & Maps, Iterators, and Algorithms

Mr. George F. Rice

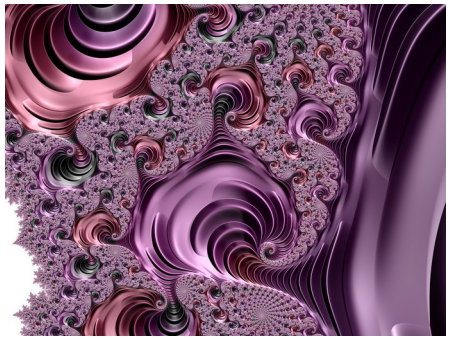
george.rice@uta.edu

Office Hours:
Prof Rice 12:30 Tuesday and Thursday in ERB 336
For TAs see this web page

You can't trust atoms.
They make up everything!

Overview: Collections, Maps, Iterators, and Algorithms

- Collections
  - Lists
  - Queues and Deques
  - Sets
- Maps
- Iterators and ListIterators
- Algorithms



All images per the Pixabay License. See credits later in this slide deck.

Review

Generic Summary

- **Generic** – A Java construct representing a method or class in terms of generic types
  - This enables the algorithm to be written independent of the types of data to which it applies
  - A type may be specified when the generic class is instanced or the generic method is called
- Potentially reusable data structures and algorithms that are suitable should generally be defined as generics

The Java Class Library (JCL)

- The JCL provides a good variety of generic data structures and algorithms focused on organizing code and data as Java classes and interfaces
  - The JCL provides an OS-neutral abstraction of the environment
  - The JCL is written mostly in Java, with hardware-dependent portions via the Java Native Interface (JNI)
  - In Java 9, the JCL was organized into modules, sometimes called the Java Module System (JMS)

Joshua Bloch, lead engineer for much of the JCL

https://docs.oracle.com/en/java/javase/16/core/java-core-libraries1.html

Photo from https://github.com/jbloch
Educational Fair Use is asserted

Commonly Used Java Modules

- **java.lang** – Provides fundamental Java classes, and **imported automatically**
- **java.util** – Defines collections, maps, iterators, algorithms, and miscellaneous utilities
- **java.io** – Provides input/output via data streams, files, and serialization
  (except Scanner, which is in java.util)
- **java.nio** – Provides for buffered (non-blocking) I/O, charsets, channels, and selectors
- **java.text** – Provides text, date, number, and messaging classes and interfaces
  (this encapsulates localization, e.g., US or Indian formats and units)
- **java.time** – Supports calculations with dates, time, duration, and instants
- **java.math** – Provides unlimited precision math types BigInteger and BigDecimal
  (don't confuse with java.lang.math that provides exponential, log, and trig functions)
- **java.net*** – Establish and communicate via network connections
- **java.security*** – Provides access control and interfaces to crypographic operations
- **javax.crypto** – Supports various symmetric, asymmetric, block, and stream cyphers
- **javax.swing** with **java.awt** – Graphical User Interface widgets and operations

* These modules also include a javax variant with additional capabilities
https://docs.oracle.com/en/java/javase/15/docs/api/java.base/module-summary.html

Java Collections Framework

- The Java Collections Framework (JCF) is defined as part of java.util and provides
  - Collections – Linear and associative data structures with supporting methods
  - Iterators – Classes that act as “smart” pointers
  - Algorithms – Polymorphic implementations that can be applied to collections
- Today we tour the JCF

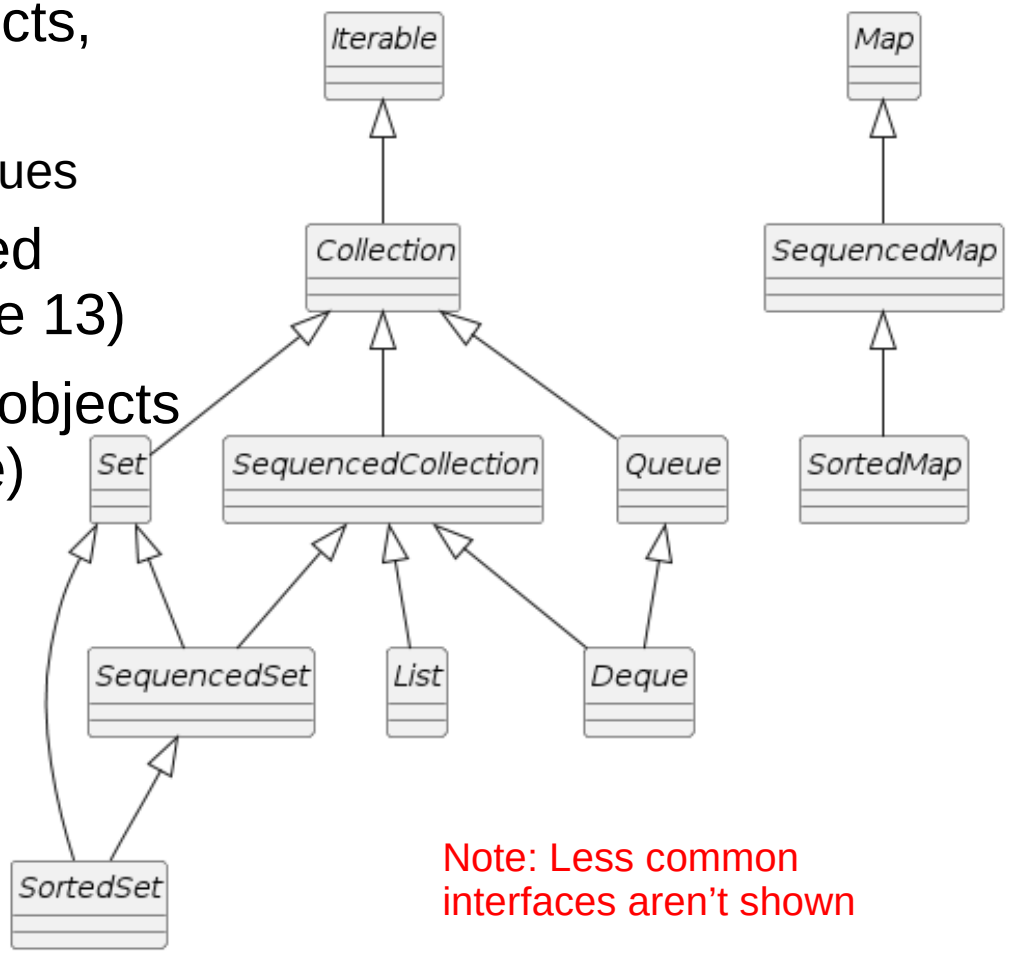
Collections and Maps

Butterfly Collection by blende12 per the Pixabay License
https://pixabay.com/photos/nature-animals-butterflies-2769471/

Map on Hands by stokpic per the Pixabay License
https://pixabay.com/photos/hands-world-map-global-earth-600497/

Simplified Interface Diagram

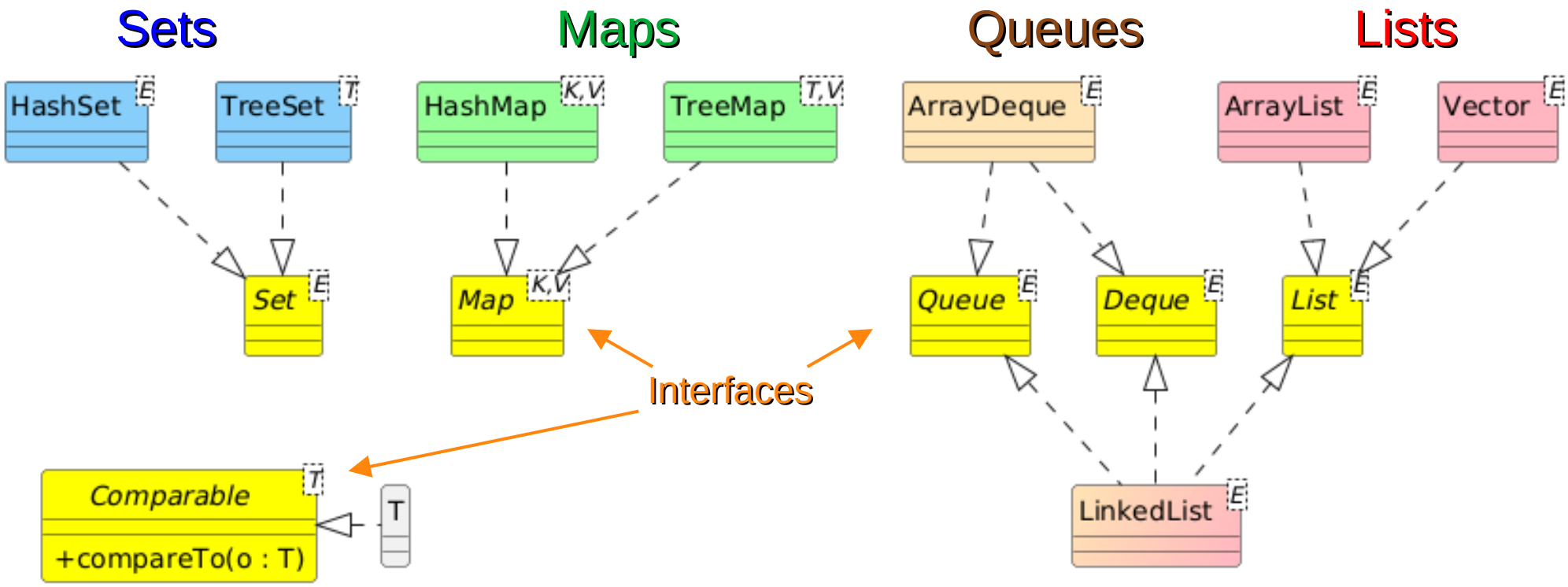
- Collection/map interfaces are *generic*
  - You can collect (almost) any objects, but not primitives
    - But all primitives have class analogues
  - You may instance with constrained types (discussed briefly in Lecture 13)
  - ArrayList<Object> can collect all objects in a single container (for example)
    - Similar to Python list and dict
- Note that Map is NOT a sub-interface of Collection
  - But is often still considered to be a type of collection conceptually



Note: Less common interfaces aren't shown

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Collection.html

Interfaces and their Implementing Classes



For our *sorted* TreeSet and TreeMap

Queue AND List

Brief Overview of Collection / Map Interfaces

- **List** stores elements with an integer index, similar to an array
  - Implemented by ArrayList, LinkedList, and Vector
- **Queue** and **Deque**[1] manage elements prior to processing in single and double ended lists, respectively
  - Both implemented by ArrayDeque and LinkedList
- **Set** is a de-duplicated collection of unique elements stored in no specific order, while SortedSet is similar but sorts the elements
  - Implemented by HashSet and **TreeSet**[2]
- **Map** is a list with an index (called the key) that is of (almost) any type, not just Integer, while SortedMap sorts the keys
  - Implemented as HashMap and **TreeMap**

* Deque (**D**ouble-**e**nded **que**ue) is pronounced “deck”

Some Classes that Implement the Collection / Map Interfaces

- We'll go over the more common collection classes that implement these interfaces
  - We'll include some example code as well
  - The Java Tutorials have more examples (and can you really ever have enough?)

Slides about collections use this tag

Slides about maps use this tag

Butterfly Collection by blende12 per the Pixabay License
https://pixabay.com/photos/nature-animals-butterflies-2769471/

Map on Hands by stokpic per the Pixabay License
https://pixabay.com/photos/hands-world-map-global-earth-600497/

Lists

- **ArrayList** is a resizable, flexible class of the standard array **optimized for appending and indexing**
  - Append (`foo.add("hi")`), insert (`foo.add(13, "hi")`), retrieve (`foo.get(13)`), and remove (`foo.remove(13)`)
  - Relatively slow insert / remove except at the end
  - Allocates more heap memory as needed
- **Vector** is just a **thread-safe** version of ArrayList
  (No, you may NOT use this for P07!)
- **LinkedList** provides a resizable, flexible version of the standard array **optimized for fast inserts / deletes**
  - Double-linked list for fast forward and reverse iteration
  - Includes push, pull, and removeLast as a queue, too!

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayList.html
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/Vector.html
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html

Lists in Action

```
import java.util.List;       // The interface
import java.util.ArrayList;  // The classes
import java.util.LinkedList; //   that implement it

public class ListExample {
    public static void main(String[] args) {
        for(List<String> list : new List[] {
                                new ArrayList<>(), new Vector<>(), new LinkedList<>()}) {
            list.add("UTA");                  // Append
            list.add("Town");                 // Append
            list.add(0, "Hello");             // Insert before UTA
            list.set(2, "World");             // Overwrite Town
            list.add("Forever!");             // Append after World
            list.remove(2);                   // Remove World
            System.out.println("Size = "
                           + list.size()      // Number of elements
               + ", UTA is index "
               + list.indexOf("UTA"));        // Search (-1 if not found)
            for(var s : list)
                System.out.print(s + " "); // Iteration
            list.clear();                     // Clear
            System.out.println("list is now "
              + (list.isEmpty() ? "empty" // isEmpty?
                                : "not empty"));
        }
    }
}
```

Similar methods for ArrayList, Vector, and LinkedList

Lists in Action

```
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$ java ListExample
Size = 3, UTA is index 1
Hello UTA Forever! list is now empty
Size = 3, UTA is index 1
Hello UTA Forever! list is now empty
Size = 3, UTA is index 1
Hello UTA Forever! list is now empty
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$
```

ArrayDeque

- **ArrayDeque** (pronounced “array deck”) is a Double-Ended QUEue (hence, “DEQUE”)
  - ArrayList is very efficient at the end, but

    ArrayDeque is very efficient at beginning OR end
  - LinkedList is also a Deque but uses more memory
- ArrayDeque is an optimized Last-In First-Out (LIFO) OR First-in First-out (FIFO) stack
  - It has no get(index) method, but it iterates
  - If you need get(index), use LinkedList instead

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ArrayDeque.html
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/LinkedList.html

LIFO / FIFO Example

```
import java.util.Deque;      // Interface
import java.util.ArrayDeque; // Class implementations
import java.util.LinkedList;

public class DequeExample {
    public static void main(String[] args) {
        Deque<Integer> lifo = new ArrayDeque<>(); // Last-In First-Out Stack
        Deque<Integer> fifo = new LinkedList<>(); // First-In First-Out Stack
        int popped;

        // Pushing is the same for LIFO and FIFO
        System.out.print("Pushing ");
        for (int i=1; i<10; ++i) {
            System.out.print("... " + i);
            lifo.push(i);
            fifo.push(i);
        }
        System.out.println('\n');

        // To pop the LIFO, use pop() method
        for(int i=0; i<3; ++i)
            System.out.println("Popped from LIFO: " + lifo.pop());
        System.out.println("LIFO is now " + lifo + '\n');

        // To pop the FIFO, use removeLast() method
        for(int i=0; i<3; ++i)
            System.out.println("Popped from FIFO: " + fifo.removeLast());
        System.out.println("FIFO is now " + fifo + '\n');
    }
}
```

Note that ArrayDeque OR LinkedList could be used for the lifo OR the fifo.

The difference is in the “pop” method:

- lifo uses pop() or removeFirst()
- fifo uses removeLast()

ArrayDeque LIFO / FIFO Example

```
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$ java ArrayDequeExample
Pushing ... 1... 2... 3... 4... 5... 6... 7... 8... 9

Popped from LIFO: 9
Popped from LIFO: 8
Popped from LIFO: 7
LIFO is now [6, 5, 4, 3, 2, 1]

Popped from FIFO: 1
Popped from FIFO: 2
Popped from FIFO: 3
FIFO is now [9, 8, 7, 6, 5, 4]

ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$
```

HashSet & TreeSet

- **HashSet** is a collection of unsorted keys, while **TreeSet** is a collection of sorted keys
  - Essentially an ArrayList of objects with **duplicates automatically removed**, and (for TreeSet) always **sorted**
  - Objects stored in HashSet / TreeSet MUST override **`hashCode`**!
  - For TreeSet, an implementation of Comparator may be provided as a constructor parameter to specify sort order
- If YOU wrote the class being used as the key (index), **you *must* define its equals() and hashCode() methods**
  - See Lecture 12 for help
- TreeSet makes a decent de-duplicated prioritized queue

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashSet.html
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeSet.html

HashSet & TreeSet Example

```java
import java.util.HashSet;
import java.util.TreeSet;
import java.util.Scanner;

public class SetExample {
    public static void main(String[] args) {
        Set<String> words = new HashSet<>();
        Set<String> sortedWords = new TreeSet<>();
        Scanner in = new Scanner(System.in);

        System.out.print("Enter a sentence: ");
        while(in.hasNext()) {
            String s = in.next();
            words.add(s);
            sortedWords.add(s);
        }
        System.out.print("Words: ");
        for(String s : words) System.out.print(s + " ");
        System.out.print("\nSorted: ");
        for(String s : sortedWords) System.out.print(s + " ");
        System.out.println("");
    }
}
```

String implements Comparable<String> and overrides hashCode(). We're good!

Module java.base
Package java.lang

Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>, C

```
public final class String
extends Object
implements Serializable, Comparable<String>, Ch
```

```
int                    hashCode()
```

HashSet & TreeSet Example

```
ricegf@antares:~/dev/202108/20$ java SetExample
Enter a sentence: Now is the time for all good men to come to the aid of their country
Words: all country for their is come good the Now men of time to aid
Sorted: Now aid all come country for good is men of the their time to
ricegf@antares:~/dev/202108/20$
```

TreeSet with Custom `Student` Class

- If we write our own class to store in a TreeSet, we must implement Comparable and override hashCode.

```
class Student implements Comparable<Student> {
    private String firstName;
    private String lastName;
    private Integer ID;
    private ArrayList<Double> grades;

    public Student(String firstName, String lastName, Integer ID) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.ID = ID;
        this.grades = new ArrayList<>();
    }

    public void addGrades(Double... grades) { // grades is an array
        for(Double grade : grades) this.grades.add(grade);
    }

    public Double[] grades() {
        return grades.toArray(new Double[grades.size()]);
    }
```

We have 4 fields for our Student class, but only the first 3 are relevant in comparisons.

`Double… grades` (called a vararg or variable argument list) accepts any number of Double parameters, loading them all into the Double[ ] grades array.

This duplicates the ArrayList into a simple array, useful for returning all data without breaking encapsulation!

Custom `Student` Class Implementations and Overrides

```
    @Override // Comparable interface
    public int compareTo(Student s) {
        int result = lastName.compareTo(s.lastName);
        if(result == 0) result = firstName.compareTo(s.firstName);
        if(result == 0) result = ID.compareTo(s.ID);
        return result;
    }
    @Override
    public boolean equals(Object o) {
        if(this == o) return true;      // Is it me?
        if(o == null                    // Is it my type?
          || ! (o instanceof Student)) return false;
        Student s = (Student) o;        // Downcast to my type
        return compareTo(s) == 0;       // Compare significant fields using compareTo
    }
    @Override
    public int hashCode() {          // If s1.equals(s2), they MUST have same hashCode.
        return Objects.hash(lastName, firstName, ID); // So use SAME fields as equals!
    }
    @Override
    public String toString() {
        double sum = 0;
        for(double grade : grades) sum += grade;
        return String.format("%s %s (%d, %3.1f average)",
                firstName, lastName, ID, sum / grades.size());
    }
}
```

Implementing Comparable enables TreeSet to sort our Student objects by lastName, firstName, and finally ID.

Our traditional equals and hashCode implementations

Creating and Using the TreeSet

- Implement and use our TreeSet<Student>

```
public class SetHashCodeExample {
    public static void main(String[] args) {
        TreeSet<Student> students = new TreeSet<>();

        Student s = new Student("Fred", "Flintstone", 1002391);
        s.addGrades(82.0, 91.0, 66.0, 102.0, 98.0);
        students.add(s);
        s = new Student("Barney", "Rubble", 1001134);
        s.addGrades(58.0, 62.0, 71.0, 59.0, 91.0, 88.0, 89.0, 76.0);
        students.add(s);
        s = new Student("Wilma", "Flintstone", 1001912);
        s.addGrades(91.0, 93.0, 88.0, 90.0);
        students.add(s);
        s = new Student("Betty", "Rubble", 1002201);
        s.addGrades(92.0, 103.0, 98.0, 101.0);
        students.add(s);

        System.out.println("Students: ");
        for(Student student : students) System.out.println("  " + student);
        System.out.println("");
    }
}
```

We create 4 instances and add them to the TreeSet.

We expect the TreeSet to iterate in the order defined by Student.compareTo!

Creating and Using the TreeSet

```
Students:
  Fred Flintstone (1002391, 87.8 average)
  Wilma Flintstone (1001912, 90.5 average)
  Barney Rubble (1001134, 74.3 average)
  Betty Rubble (1002201, 98.5 average)
```

HashMap & TreeMap

- **HashMap** is a collection of key-value pairs in any order, **TreeMap** is the same but sorted by key
  - Essentially an ArrayList of objects with (almost) any type as the key (index)
  - Keys in HashMap / TreeMap MUST override **`hashCode`**!
  - For TreeMap, an implementation of Comparator for key may be provided as a constructor parameter to specify sort order
- If YOU wrote the class being used as the key (index), **you *must* define equals() and hashCode() methods**
  - See Lecture 12 for help
  - Note: enigma/Setting.java was the key of a TreeMap to sort the encrypted strings in input.txt by speed (based on my custom compareTo implementation)

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/HashMap.html
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/TreeMap.html

HashMap & TreeMap Example

- Class coordinate stores lat-longs around the globe
- BEWARE: If your compareTo method returns 0 (“equals”) then the new element will *overwrite* an existing Map entry!
  - That’s why compareTo also checks longitude

```
class Coordinate implements Comparable<Coordinate> {
    public Coordinate(Degrees latitude, Degrees longitude) {
        this.latitude = latitude;
        this.longitude = longitude;
    }
    @Override
    public String toString() {
        return String.format("(%s, %s)", latitude, longitude);
    }
    @Override
    public int compareTo(Coordinate c) {
        int result = latitude.getDegrees().compareTo(c.latitude.getDegrees());
        if(result == 0)
            result = longitude.getDegrees().compareTo(c.longitude.getDegrees());
        return result;
    }
    // Remaining code omitted
```

Degrees stores a latitude or longitude.
See TreasureMap.java for ALL of the code

HashMap & TreeMap Example

```
public class TreasureMap {
  public static void main(String[] args) {
    Map<Coordinate, String> unsortedTreasures = new HashMap<>();
    Map<Coordinate, String>   sortedTreasures = new TreeMap<>();

    Coordinate c2 = new Coordinate(new Degrees(30.6266, Direction.N),
                                   new Degrees(81.4609, Direction.W));
    unsortedTreasures.put(c2, "Treasure of San Miguel");
      sortedTreasures.put(c2, "Treasure of San Miguel");

    Coordinate c1 = new Coordinate(new Degrees(5.5282, Direction.N),
                                   new Degrees(87.0574, Direction.W));
    unsortedTreasures.put(c1, "Treasure of Lima");
      sortedTreasures.put(c1, "Treasure of Lima");

    Coordinate c3 = new Coordinate(new Degrees(60.28889, Direction.S),
                                   new Degrees(19.04444, Direction.E));
    unsortedTreasures.put(c3, "Treasure Island");
      sortedTreasures.put(c3, "Treasure Island");

    System.out.println("Unsorted treasures: ");
    for(Coordinate key : unsortedTreasures.keySet()) {
        System.out.println("    " + unsortedTreasures.get(key) + " " + key);
    }
    System.out.println("Sorted (by latitude) treasures: ");
    for(Coordinate key : sortedTreasures.keySet()) {
        System.out.println("    " + sortedTreasures.get(key) + " " + key);
    }
  }
}
```

HashMap doesn't sort, TreeMap does!

Finding Treasure Island

```
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$ java TreasureMap
Unsorted treasures:
    Treasure Island (60.2889 S, 19.0444 E)
    Treasure of San Miguel (30.6266 N, 81.4609 W)
    Treasure of Lima (5.5282 N, 87.0574 W)
Sorted (by latitude) treasures:
    Treasure of Lima (5.5282 N, 87.0574 W)
    Treasure of San Miguel (30.6266 N, 81.4609 W)
    Treasure Island (60.2889 S, 19.0444 E)
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$
```

Hashless Treasure
(What Becomes of Skipping hashCode)

EqualsAndHashCode.java

```
class HashlessTreasure {
    public HashlessTreasure(Coordinate c, String name, double value) {
        this.coordinate = c;
        this.treasureName = name;
        this.treasureValue = value;
    }

    @Override
    public boolean equals(Object o) {
        if(this == o) return true;
        if(o == null || this.getClass() != o.getClass()) return false;
        HashlessTreasure t = (HashlessTreasure) o; // Downcast to a HashlessTreasure
        return coordinate.equals(t.coordinate)
            && treasureName.equals(t.treasureName)
            && (treasureValue == t.treasureValue);
    }
    protected Coordinate coordinate;  // Our custom class
    protected String treasureName;    // A JCL class
    protected double treasureValue;   // A primitive
}
```

For this example class, assume Coordinate has already defined equals() and hashCode() (see code on cse1325-prof/15).

Here we skip defining hashCode(). Uh-oh...

HashMap Example

EqualsAndHashCode.java

```
class Treasure extends HashlessTreasure {
    public Treasure(Coordinate c, String name, double value) {
        super(c, name, value);
    }
    @Override
    public int hashCode() {
        return Objects.hash(coordinate, treasureName, treasureValue);
    }
}
```

We add a proper hashCode in the subclass.

HashMap Example

Now we try both classes with a Map. What could possibly go wrong?

```
public class EqualsAndHashCode {
    public static void main(String[] args) {
        HashMap<HashlessTreasure, Integer> map1 = new HashMap<>();
        map1.put(new HashlessTreasure(new Coordinate(23,17), "Blackbeard's", 12835.19), 0);
        map1.put(new HashlessTreasure(new Coordinate(23,17), "Blackbeard's", 12835.19), 1);
        System.out.println("Without hashCode, this should (incorrectly) be 2: "
                                                                + map1.size());

        HashMap<HashlessTreasure, Integer> map2 = new HashMap<>();
        map2.put(new Treasure(new Coordinate(23,17), "Blackbeard's", 12835.19), 2);
        map2.put(new Treasure(new Coordinate(23,17), "Blackbeard's", 12835.19), 3);
        System.out.println("WITH    hashCode, this should (correctly)   be 1: "
                                                                    + map2.size());
    }
}
```

Which type's hashCode() will be called? Why?

This Story's Moral: Don't forget your equals *and hashMap* implementations!

```
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$ java EqualsAndHashCode
Without hashCode, this should (incorrectly) be 2: 2
WITH    hashCode, this should (correctly)   be 1: 1
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$
```

For-Each and Map

- Map doesn’t work with for-each loops like Collections
  - But we can obtain a Set of either keys or values that DO work with for-each loops!
    - **keySet()** returns a Set of keys, which can also be used as subscripts to the Map object via method get
    - **valueSet()** returns a Set of values from the Map
- Given any map, then, this prints the key-value pairs

```
for (var key: map.keySet()) {
    System.out.println(key + ":" + map.get(key));
}
```

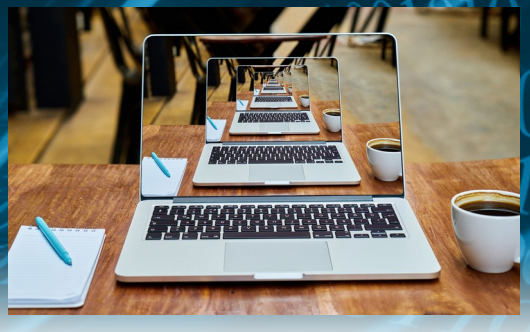
Collection and Map
Common Methods to Know!

- Add an Element
  - `add(E e)` – List, Deque, Set (To end)
  - `push(E e)` – Deque (Insert at 0)
  - `put(K key, E value)` – Map
- Get an Element
  - `E get(int index)` – List
  - `V get(Object key)` – Map
- Remove an Element
  - `remove(int index)` – List
  - `E removeLast()` – List, Deque (From end)
  - `E pop()` – Deque (From index 0)
  - `boolean remove(Object key)` – List, Deque, Set, Map
  - `clear()` – List, Deque, Set, Map
- Check the Number of Elements
  - `int size()` – List, Deque, Set, Map
  - `boolean isEmpty()` – List, Deque, Set, Map
- Copy to Array
  - `Object[] toArray()` – List, Deque, Set
  - `T[] toArray(T[] a)` – List, Deque, Set
- Search
  - `int indexOf(Object o)` – List
  - `boolean contains(Object o)` – List, Deque, Set
  - `boolean containsKey(Object o),`
    `boolean constainsValue(Object o)` – Map
- Iterate
  - `for(var v : vs)` – List, Set
  - `for(var key : map.keySet())` – Map
    `var value = map.get(key);`

Iterators

Iterated Laptops by geralt per the Pixabay License
https://pixabay.com/illustrations/laptop-technology-online-monitor-6962810/

Iterator (Interface)

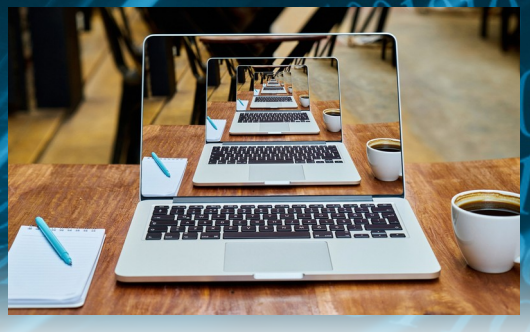


- **Iterator**: A pointer-like object used to access items managed by a Collection
  - Obtain an iterator with the collection's **iterator()** method
- The iterator has 3 key methods
  - **hasNext()** method returns true if another element is available
  - **next()** method returns the next element and advances
  - **remove()** method removes the last element returned by next()

This is quite reminiscent of BufferedReader, yes?

https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Iterator.html

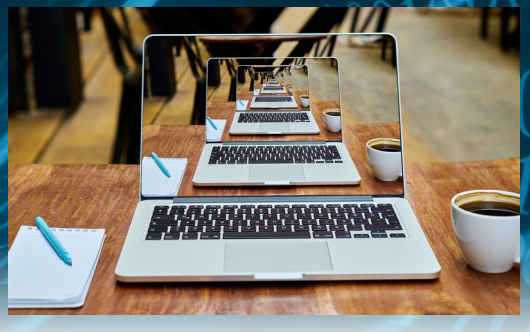
Simple Iterator Example



- Class Food has ArrayLists of Fruits and Veggies
  - It returns an Iterator for each with iFruit() and iVeggie()

```
class Fruit { // and Veggie is similar
    public Fruit(String name) {this.name = name;}
    // Also overrides toString, equals, and hashCode
    private String name;
}

public class Food {
    // Lists of foods
    private ArrayList<Fruit> fruits = new ArrayList<>(
        Arrays.asList(new Fruit("Apple"), new Fruit("Pear"), new Fruit("Grape")));
    private ArrayList<Veggie> veggies = new ArrayList<>(
        Arrays.asList(new Veggie("Carrot"), new Veggie("Pea"), new Veggie("Tomato")));

    // Iterators for the lists
    public Iterator<Fruit>  iFruit()  {return fruits.iterator();}
    public Iterator<Veggie> iVeggie() {return veggies.iterator();}
}
```

Simple Iterator Example



- With an Iterator we have access to the contents of each ArrayList

```
import java.util.Iterator;

public class SimpleIterator {
    public static void main(String[] args) {
        Food food = new Food();
        System.out.println("Fruits include ");

        // Iterator is generic, and starts by “pointing to” the first element
        Iterator<Fruit> it = food.iFruit();

        // hasNext() is true if it “points to” an object
        // next() returns the object and increments the iterator
        while(it.hasNext()) System.out.println("  " + it.next());
    }
}
```

```
ricegf@antares:~/dev/202301/23/code_from_slides$ java SimpleIterator
Fruits include
  Fruit Apple
  Fruit Pear
  Fruit Grape
ricegf@antares:~/dev/202301/23/code_from_slides$
```

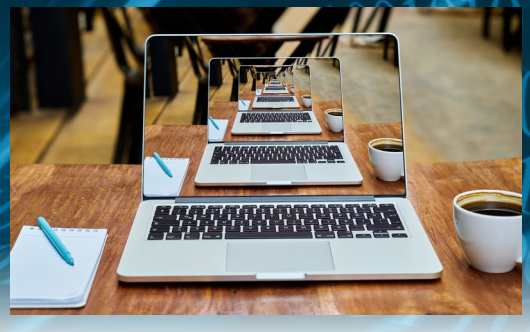
Another Iterator Example

- Iterator is an interface, thus Iterator may be a variable, parameter, or return type

```
import java.util.Iterator;

public class IteratorExample {
    private static void printIterator(Iterator it) { // Any cast of Iterator works
        String sep = "";
        while(it.hasNext()) {
            System.out.print(sep + it.next());
            sep = ", ";
        }
    }
    public static void main(String[] args) {
        Food food = new Food();

        System.out.print("Fruits include ");
        printIterator(food.iFruit());

        System.out.print("\nVeggies include ");
        printIterator(food.iVeggie());
        System.out.println("");
    }
}
```

```
ricegf@antares:~/dev/202301/23/code_from_slides$ javac Itera
ricegf@antares:~/dev/202301/23/code_from_slides$ java Iterat
Fruits include Fruit Apple, Fruit Pear, Fruit Grape
Veggies include Veggie Carrot, Veggie Pea, Veggie Tomato
ricegf@antares:~/dev/202301/23/code_from_slides$
```

ListIterator



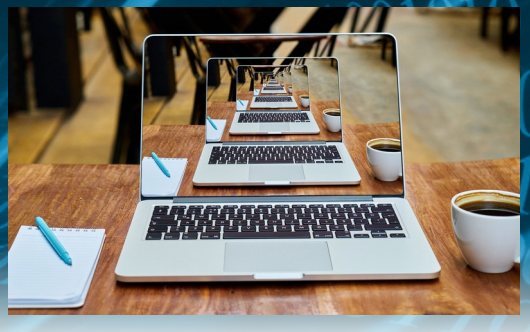
- A ListIterator is a bi-directional Iterator
  - Obtain with the listIterator() method on a supporting collection

`Iterator<E>`	`iterator()`	Returns an iterator over the elements in this list in proper sequence.
`int`	`lastIndexOf(Object o)`	Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the elem
`ListIterator<E>`	`listIterator()`	Returns a list iterator over the elements in this list (in proper sequence).
`ListIterator<E>`	`listIterator(int index)`	Returns a list iterator over the elements in this list (in proper sequence), starting at the specified position in the list.

- The iterator adds 6 more capable methods to Iterator
  - **hasPrevious()** method returns true if the previous element is available
  - **previous()** method returns the next element
  - **nextIndex()** and **previousIndex()** returns the index of the element to which the ListIterator points
  - **add(E e)** inserts the element into the collection at the index to which the ListIterator points
  - **set(E e)** overwrites the element to which ListIterator points (but only if neither add nor remove have been called yet)

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ListIterator.html

ListIterator Example



- Add ListIterator getters to class Food
  - You don't really need Iterators *and* ListIterators
  - Iterator is ListIterator's superclass (superinterface?)

```
class Fruit { // and Veggie is similar
    public Fruit(String name) {this.name = name;}
    // Also overrides toString, equals, and hashCode
    private String name;
}

public class Food {
    // Lists of foods
    private List<Fruit> fruits = new ArrayList<>(
        Arrays.asList(new Fruit("Apple"), new Fruit("Pear"), new Fruit("Grape")));
    private List<Veggie> veggies = new ArrayList<>(
        Arrays.asList(new Veggie("Carrot"), new Veggie("Pea"), new Veggie("Tomato")));

    // Iterators for the lists
    public Iterator<Fruit>  iFruit()  {return fruits.iterator();}
    public Iterator<Veggie> iVeggie() {return veggies.iterator();}
    public ListIterator<Fruit> liFruit() {return fruits.listIterator();}
    public ListIterator<Veggie> liVeggie() {return veggies.listIterator();}
}
```

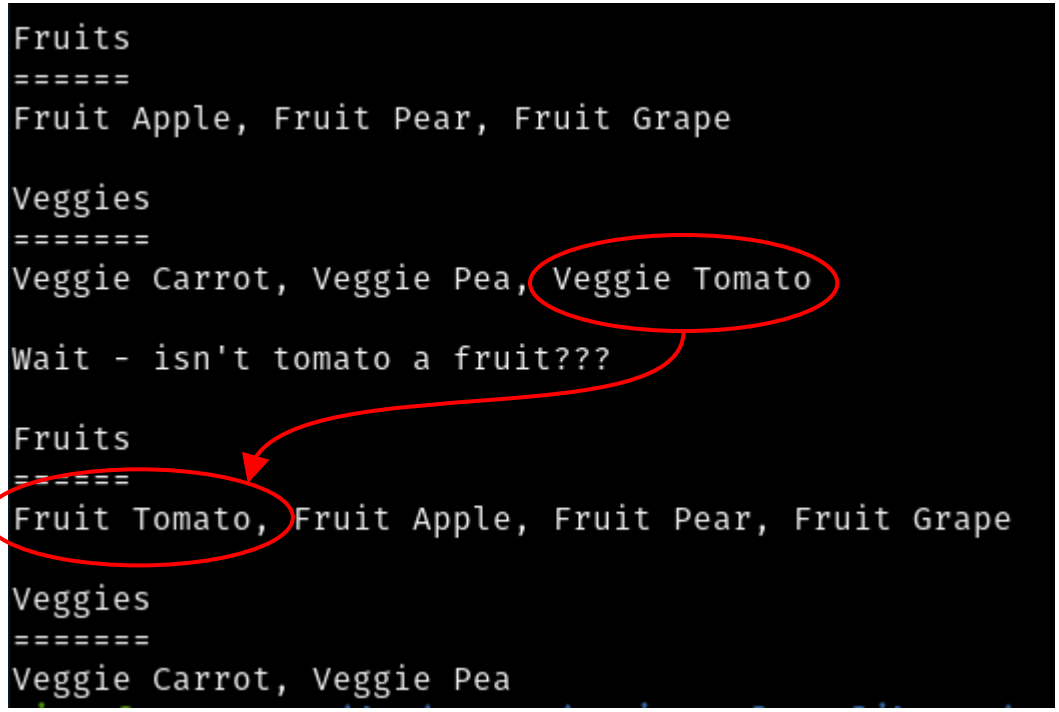
A ListIterator Example

- Is tomato a fruit or a vegetable?
  - Let's switch ArrayLists!

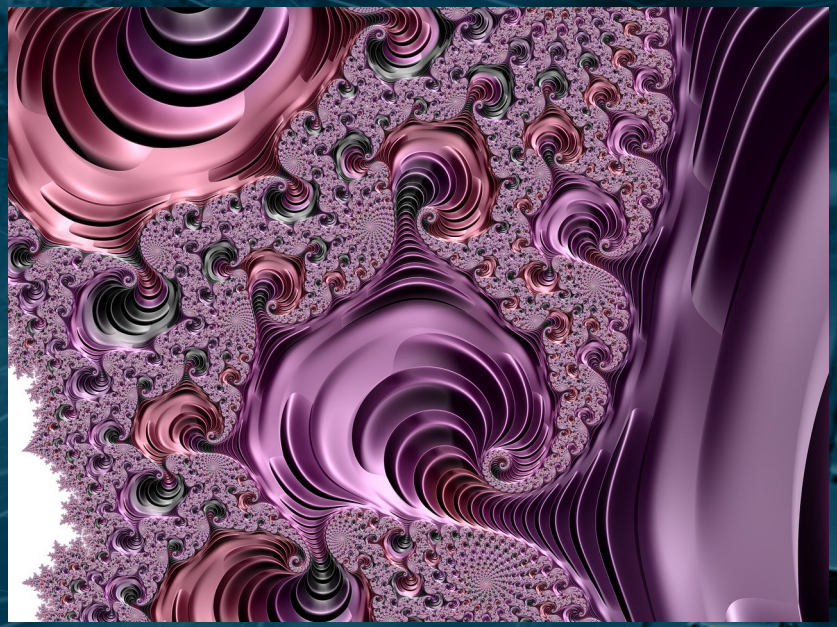
Move tomato to fruit!

```
public static void main(String[] args) {
    Food food = new Food();
    printIterator("Fruits", food.liFruit());
    printIterator("Veggies", food.liVeggie());
    System.out.println("\nWait - isn't tomato a fruit???\n");

    Veggie tomato = new Veggie("Tomato");    // Delete all Veggie("Tomato")
    ListIterator<Veggie> vi=food.liVeggie(); // Iterate through the veggies
    while(vi.hasNext()) if(vi.next().equals(tomato)) vi.remove();
    ListIterator<Fruit> fi = food.liFruit(); // Point to start of fruits
    fi.add(new Fruit("Tomato"));             // Insert Fruit("Tomato") at start

    printIterator("Fruits", food.liFruit());
    printIterator("Veggies", food.liVeggie());
}
```

A ListIterator Example



Algorithms



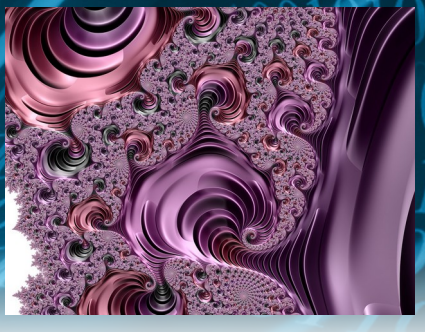
Abstract Art by Patty Talavera per the Pixabay License
https://pixabay.com/illustrations/abstract-art-fractal-art-fractal-1476001/

Collections: A Class of Algorithms

- An **Algorithm** is a procedure for solving a specific problem, expressed in terms of an ordered set of actions to execute
  - Sorting, shuffling, searching, and analyzing are common algorithms
- For implementers of the Collection interface, these algorithms are found as static methods in the **Collections** class

https://docs.oracle.com/en/java/javase/16/docs/api/java.base/java/util/Collections.html

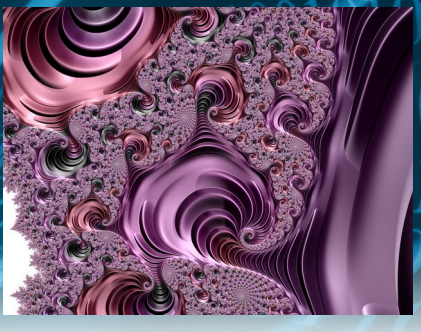
Sort



- Collections.**sort** accepts a Collection (and optional Comparator) and sorts in place
  - Quicksort is used for primitives, merge sort for objects

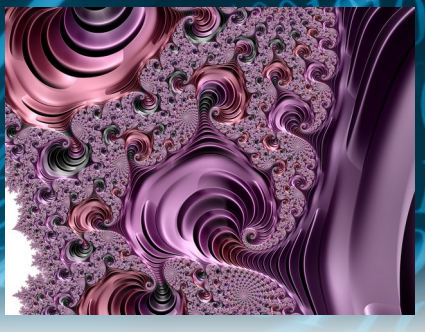
```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Sort {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        for(String s : args) list.add(s);
        Collections.sort(list);
        for(String s : list) System.out.print(s + " ");
        System.out.println("");
    }
}
```

Sort



```
ricegf@antares:~/dev/202108/20$ javac Sort.java
ricegf@antares:~/dev/202108/20$ java Sort When in the course of human events
When course events human in of the
ricegf@antares:~/dev/202108/20$
```

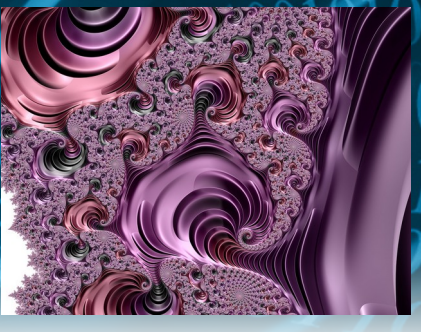
Shuffle



- Collections.shuffle accepts a Collection (and optional Random) and randomizes element order
  - This is useful for statistical analysis and, uh, games

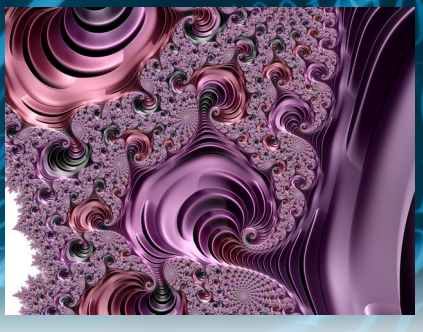
```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Shuffle {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        for(String s : args) list.add(s);
        Collections.shuffle(list);
        for(String s : list) System.out.print(s + " ");
        System.out.println("");
    }
}
```

Shuffle



```
ricegf@antares:~/dev/202108/20$ javac Shuffle.java
ricegf@antares:~/dev/202108/20$ java Shuffle When in the course of human events
in the of human course events When
ricegf@antares:~/dev/202108/20$ java Shuffle When in the course of human events
human events course When of the in
ricegf@antares:~/dev/202108/20$
```

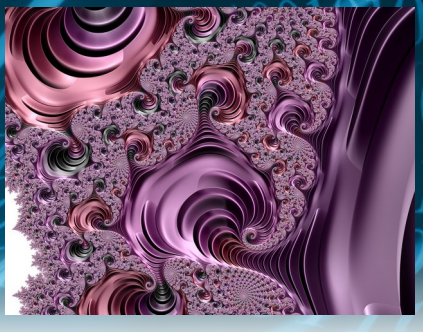
Reverse



- Collections.reverse accepts a Collection and reverses element order

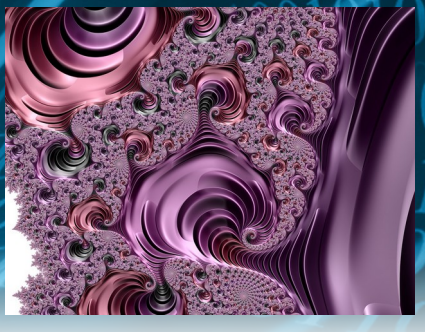
```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Reverse {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        for(String s : args) list.add(s);
        Collections.reverse(list);
        for(String s : list) System.out.print(s + " ");
        System.out.println("");
    }
}
```

Reverse



```
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$ java Reverse
Now is the time for all good men to come to the aid of their country
country their of aid the to come to men good all for time the is Now
ricegf@antares:~/dev/202408/15-java-class-library/code_from_slides$
```

Fill, Copy

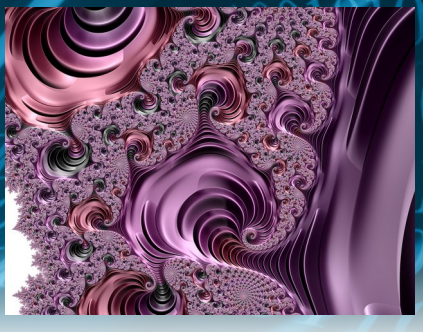


- Collections.fill sets each element to the specified value
- Collections.copy overwrites the first collection with the second collection

```
import java.util.List;
import java.util.Arrays;
import java.util.ArrayList;
import java.util.Collections;

public class FillCopy {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);  // convert array to List
        List<String> etc = new ArrayList<>(list); // construct copy of list
        Collections.fill(etc, "etc");             // fill etc with "etc"
        Collections.copy(etc, list.subList(0,list.size()/2)); // copy half of list
        for(String s : etc) System.out.print(s + " ");
        System.out.println("");
    }
}
```

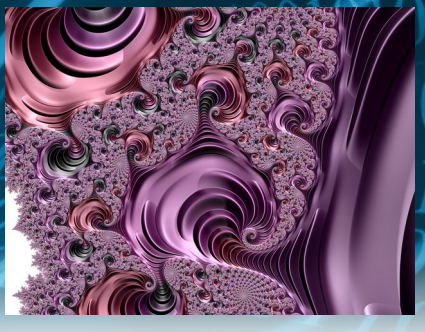
subList returns a view of THIS list between the indices as (i1, i2] (1st index included, 2nd excluded)

Fill, Copy



```
ricegf@antares:~/dev/202108/20$ javac FillCopy.java
ricegf@antares:~/dev/202108/20$ java FillCopy When in the course of human events it becomes necessary
When in the course of etc etc etc etc etc
ricegf@antares:~/dev/202108/20$
```

binarySearch

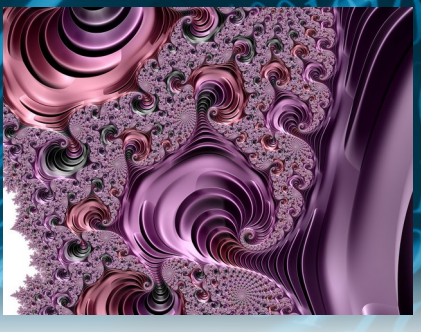


- Collections.binarySearch returns the index of one of the elements matching the key (with optional Comparator)

```
public class Search {
    public static void main(String[] args) {
        if(args.length == 0) System.err.println("Provide some arguments to search!");
        List<String> list = Arrays.asList(args); // convert array to List

        Collections.sort(list); // binary searches only work on sorted lists
        System.out.println("Your arguments have been sorted to support binarySearch!");

        for(String s : list) System.out.print(s + " ");
        System.out.println();
        for(int i=0; i<list.size(); ++i)
            System.out.print("" + i + " ".repeat(list.get(i).length()));
        System.out.println();

        Scanner in = new Scanner(System.in);
        while(true) {
            System.out.print("Search key: ");
            String key = in.nextLine(); if(key.isEmpty()) break;
            int index = Collections.binarySearch(list, key);
            if(index >= 0) System.out.println(key + " found at index " + index);
            else System.out.println(key + " not found");
        }
    }
}
```

Note that binary searches ONLY work on sorted lists!

binarySearch



```
Your arguments have been sorted to support binarySearch!
Peter Piper a of peck peppers picked pickled
0     1     2 3  4    5       6      7
Search key: job
job not found
Search key: picked
picked found at index 6
Search key: pickled
pickled found at index 7
Search key: pick
pick not found
Search key: Piper
Piper found at index 1
Search key: 0
0 not found
Search key:
```

Java Collections Framework Summary

- We briefly explored the 3 types of JCF resources
- **Collections** and **Maps** for managing data
  - lists, deques, sets, and maps
- **Iterator** and **ListIterator** for manipulating data
- A few of the almost 70 **Algorithms** in Collections
  - sort, binarySearch, reverse, shuffle, fill, copy, and more!