CSE 1325: Object-Oriented Programming

Lecture 16

Writing a Concurrent OOP Java Program

Exam #2 Review

Mr. George F. Rice
george.rice@uta.edu

Office Hours:
Prof Rice 11:00 Tuesday and Thursday in ERB 336
For TAs see this web page

A calendar's days are numbered.

Overview: Concurrency

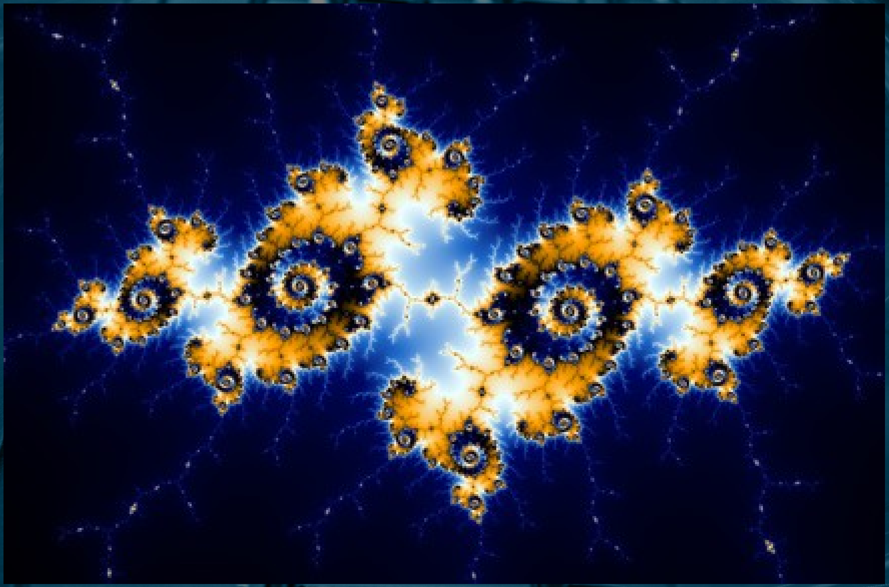
- Brief Review of Java Threads
- Mandelbrot Theory
- Mandelbrot Implementation
  - Complex class
  - Single vs Fixed Threads
  - Filters and Scrolling
  - Thread Pools
  - Swing User Interface (Demo Only)

Concurrent OOP App

- We traditionally built an app before each exam
  - But this app doesn't illustrate the techniques as well as Ralph the Robot did for the first exam
- But here's the code, anyway
- The Mandelbrot set is the set of complex numbers c for which the function $f_c(z)=z^2+c$ does NOT diverge to infinity when iterated from z=0
- Plotting x+yi as (x,y) with the color as the number of iterations before exceeding an
  arbitrary bound results is a *fractal* curve
- A fractal curve retains its intrinsic irregular shape regardless of magnification, and seems to resemble nature in uncanny ways

The Mandelbrot Set

- We’ll need (though not strictly require) the ability to handle complex numbers
  - Java does NOT provide this
- Then we can write a Mandelbrot class to generate the images
  - Happily, each pixel can be **independently** calculated – which provides an excellent opportunity for concurrency!



Code is on GitHub!

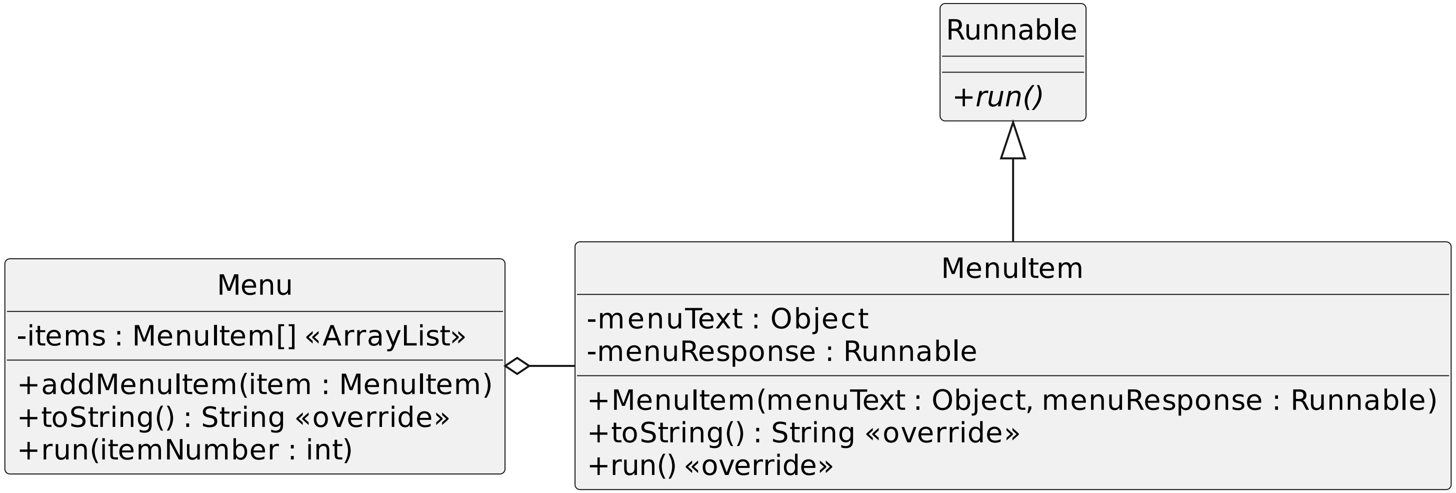
- You may explore the final Java version of this simple menu driven interface program
- A similar C++ version is on GitHub for your exploration and experimentation as we move to that language after the exam

Additional Review

- Here are the topics likely to be seen on the exam, with some code to stimulate questions

Menu and MenuItem Classes

- **`Menu`** is an encapsulated ArrayList of **`MenuItem`** (or **`? Extends Runnable`**) objects
  - Its toString() method prints each MenuItem’s index and text
  - A **`run`** method that calls the associated method using its index
- This is a more object-oriented (and more maintainable) approach to a menu-driven interface



Runnable

+run()

Menu

-items : MenuItem[] «ArrayList»

+addMenuItem(item : MenuItem)
+toString() : String «override»
+run(itemNumber : int)

MenuItem

-menuText : Object
-menuResponse : Runnable

+MenuItem(menuText : Object, menuResponse : Runnable)
+toString() : String «override»
+run() «override»

Menu.toString()
prints the menu! →

Syncing Menu and Dispatch

- The constructor builds the Menu as a simple, *executable* table – *much* easier to maintain!

```
public class EclecticMenuItems {
    private String title;
    private String output;
    private ArrayList<Object> stuff;
    private Scanner in = new Scanner(System.in);
    private Menu menu;

    public EclecticMenuItems(String title) {
        this.title = title;
        this.stuff = new ArrayList<>();
        this.output = "";
        this.menu = new Menu();

        menu.addMenuItem(new MenuItem("Add an integer",      () -> addInt()));
        menu.addMenuItem(new MenuItem("Add a double",        () -> addDouble()));
        menu.addMenuItem(new MenuItem("Add a boolean",       () -> addBoolean()));
        menu.addMenuItem(new MenuItem("Add a char",          () -> addChar()));
        menu.addMenuItem(new MenuItem("Add a string",        () -> addString()));
        menu.addMenuItem(new MenuItem("List all items",      () -> listAllItems()));
        menu.addMenuItem(new MenuItem("Sort all items",      () -> sortAllItems()));
        menu.addMenuItem(new MenuItem("Move an item",        () -> moveItem()));
        menu.addMenuItem(new MenuItem("Swap two items",      () -> swapTwoItems()));
        menu.addMenuItem(new MenuItem("Search for an item",  () -> searchForItem()));
        menu.addMenuItem(new MenuItem("Exit",                () -> endApp()));
    }
```

The fields from the previous slide

The constructor!

The *lambda* converts a method into a Runnable object

Key Code to Know!

If they select this call this !

Now the dispatch table *looks* like a dispatch table!

A Note on
Method Reference Objects

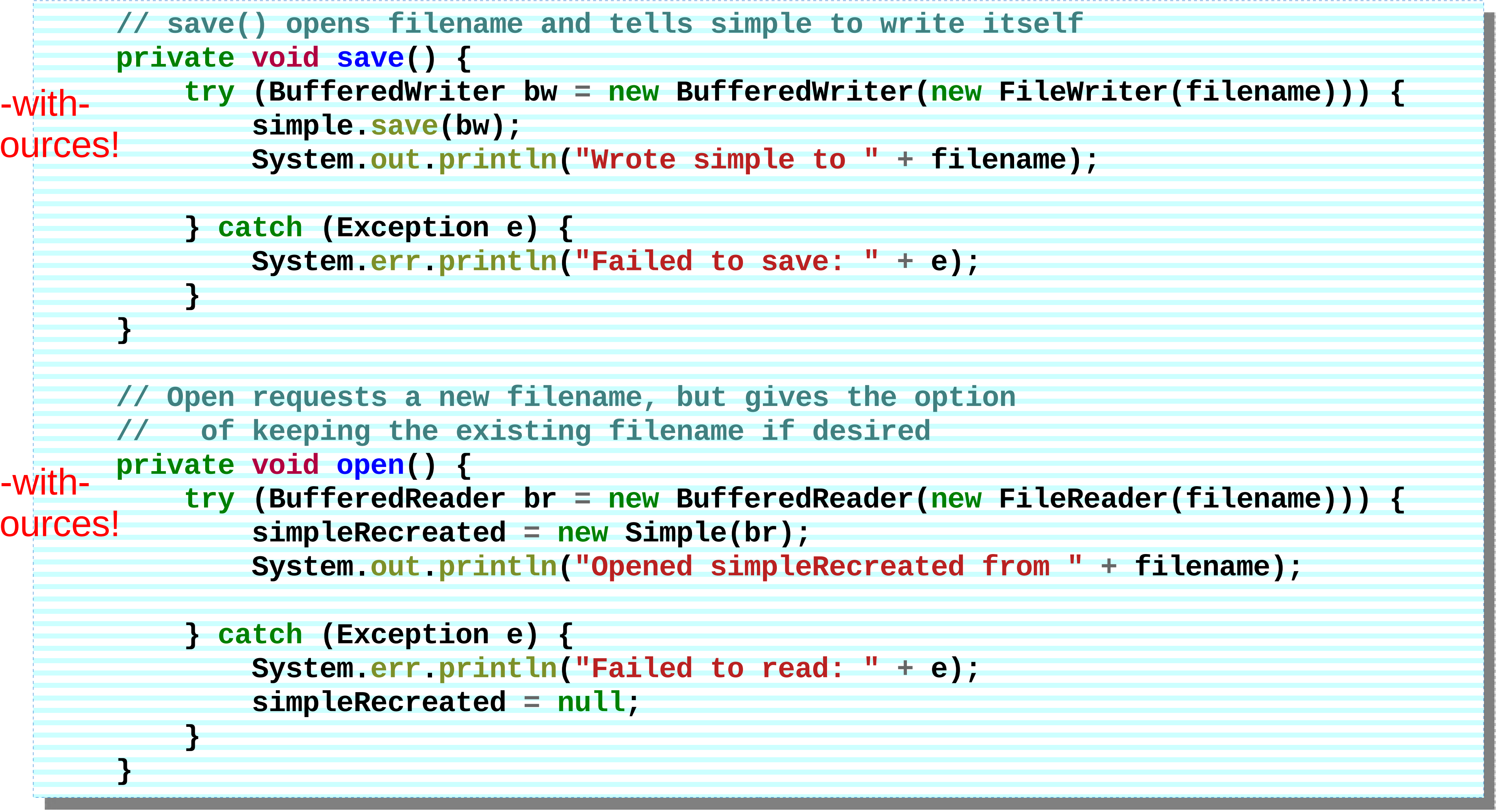
- Warning, Will Robinson!

```
public class EclecticMenuItems {
    private String title;
    private String output;
    private ArrayList<Object> stuff;
    private Scanner in = new Scanner(System.in);
    private Menu menu;

    public EclecticMenuItems(String title) {
        this.title = title;
        this.stuff = new ArrayList<>();
        this.output = "";
        this.menu = new Menu();

        menu.addMenuItem(new MenuItem("Add an integer",     this::addInt()));
        menu.addMenuItem(new MenuItem("Add a double",       () -> addDouble()));
        menu.addMenuItem(new MenuItem("Add a boolean",      () -> addBoolean()));
        menu.addMenuItem(new MenuItem("Add a char",         () -> addChar()));
        menu.addMenuItem(new MenuItem("Add a string",       () -> addString()));
        menu.addMenuItem(new MenuItem("List all items",     () -> listAllItems()));
        menu.addMenuItem(new MenuItem("Sort all items",     () -> sortAllItems()));
        menu.addMenuItem(new MenuItem("Move an item",       () -> moveItem()));
        menu.addMenuItem(new MenuItem("Swap two items",     () -> swapTwoItems()));
        menu.addMenuItem(new MenuItem("Search for an item", () -> searchForItem()));
        menu.addMenuItem(new MenuItem("Exit",               () -> endApp()));
    }
```

¼ of students used a *methed refererence object*
Instead of a *lambda* on P05. This works, but
1. We don't cover MROs in CSE1325.
2. Lambdas are much more flexible.
3. We'll expect a lambda on the exam!

Review: Saving / Opening Files

- We need save() and open() methods



```
    // save() opens filename and tells simple to write itself
    private void save() {
        try (BufferedWriter bw = new BufferedWriter(new FileWriter(filename))) {
            simple.save(bw);
            System.out.println("Wrote simple to " + filename);

        } catch (Exception e) {
            System.err.println("Failed to save: " + e);
        }
    }

    // Open requests a new filename, but gives the option
    //   of keeping the existing filename if desired
    private void open() {
        try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
            simpleRecreated = new Simple(br);
            System.out.println("Opened simpleRecreated from " + filename);

        } catch (Exception e) {
            System.err.println("Failed to read: " + e);
            simpleRecreated = null;
        }
    }
```

Try-with-resources!

Try-with-resources!

Review: Writing / Reading Data to / from Files

- Write each field on a separate line to BufferedWriter

```
public void save(BufferedWriter bw) throws IOException {
    bw.write(     aString  + '\n');
    bw.write("" + anInt    + '\n');
    bw.write("" + aDouble  + '\n');
    bw.write("" + aChar    + '\n');
    bw.write("" + aBoolean + '\n');
}
```

IMPORTANT: The order of each write and corresponding readLine must match *exactly*!

- Recreate each field from a BufferedReader line

```
public Simple(BufferedReader br) throws IOException {
    this.aString  =                      br.readLine();
    this.anInt    = Integer.parseInt    (br.readLine());
    this.aDouble  = Double.parseDouble  (br.readLine());
    this.aChar    =                      br.readLine().charAt(0);
    this.aBoolean = Boolean.parseBoolean(br.readLine());
}
```

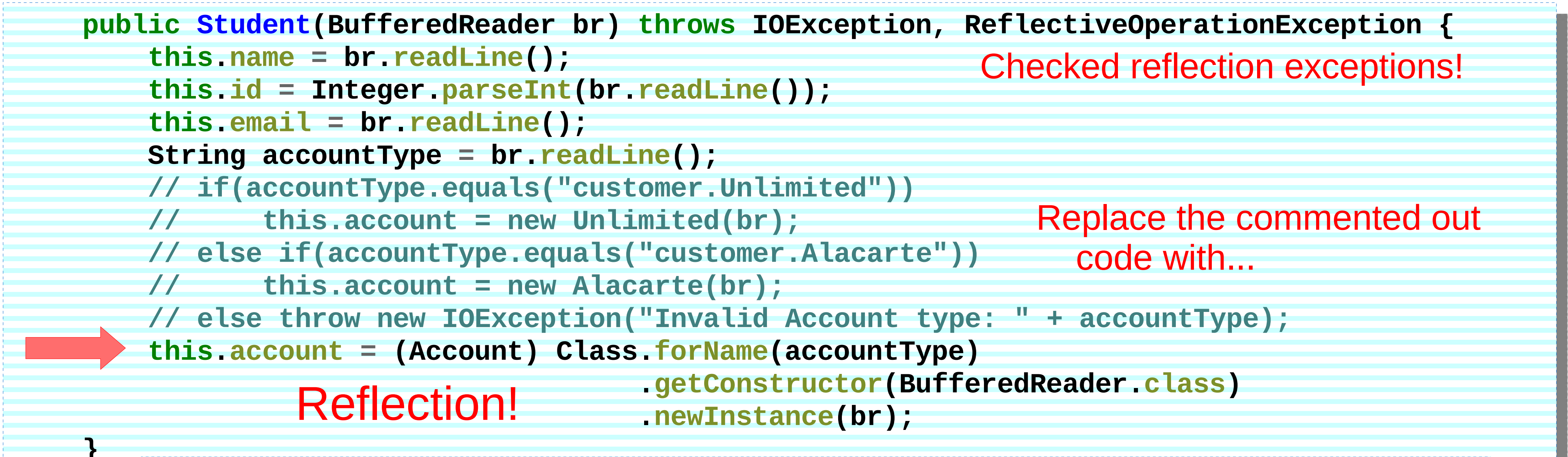
Here we elect to throw IOException out of the constructor / method.
Columns are exaggerated to emphasize the pattern to follow.

Less Simple Classes

- Enums
  - For Enum `E e;`, save as `bw.write(e.name());`
    and restore as `e = E.valueOf(br.readLine());`
- Arrays, ArrayLists, and other Collections / Maps
  - For `ArrayList<Double> ds;`, save the size first then each element:
    `bw.write(ds.size()); for(Double d: ds) bw.write(“” + d + ‘\n’);`
  - Recreate the List or Map and then add each element in turn
    `ds = new ArrayList<>(); int size = Integer.parseInt(br.readLine());`
    `while(size-- > 0) ds.add(Double.parseDouble(br.readLine());`
- Classes with fields that are classes
  - Classes we wrote should already have save methods and constructors
  - Other classes we must address individually – see their JavaDoc pages!
- Superclasses and subclasses
  - For superclass X, given `X x;`, first write subclass name, then save the object:
    `bw.write(x.getClass().getName()); x.save(bw);`
  - To restore, check the subclass name to determine the subclass constructor:
    `String s = br.readLine();if(s.equals(“pkg.SubX”)) x = new SubX(br)`;

NOT ON EXAM: Java Reflection

- Some students asked how to avoid needing to know the subclass type when restoring an Account subclass

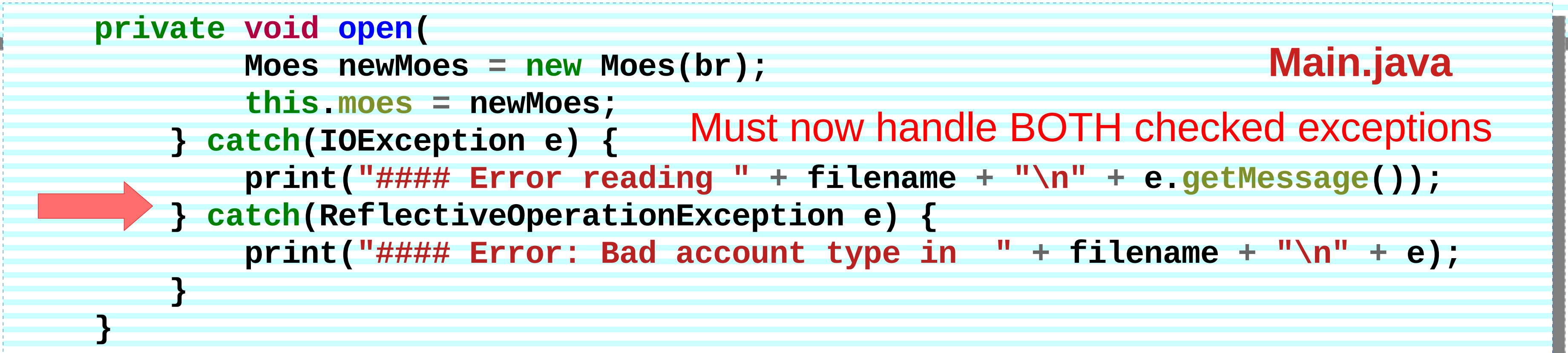


```
public Student(BufferedReader br) throws IOException, ReflectiveOperationException {
    this.name = br.readLine();
    this.id = Integer.parseInt(br.readLine());
    this.email = br.readLine();
    String accountType = br.readLine();
    // if(accountType.equals("customer.Unlimited"))
    //     this.account = new Unlimited(br);
    // else if(accountType.equals("customer.Alacarte"))
    //     this.account = new Alacarte(br);
    // else throw new IOException("Invalid Account type: " + accountType);
    this.account = (Account) Class.forName(accountType)
                                  .getConstructor(BufferedReader.class)
                                  .newInstance(br);
}
```

Checked reflection exceptions!

Replace the commented out code with...

Reflection!



```
private void open(
        Moes newMoes = new Moes(br);
        this.moes = newMoes;
    } catch(IOException e) {
        print("#### Error reading " + filename + "\n" + e.getMessage());
    } catch(ReflectiveOperationException e) {
        print("#### Error: Bad account type in  " + filename + "\n" + e);
    }
}
```

Main.java

Must now handle BOTH checked exceptions

NOT ON EXAM: Java Reflection

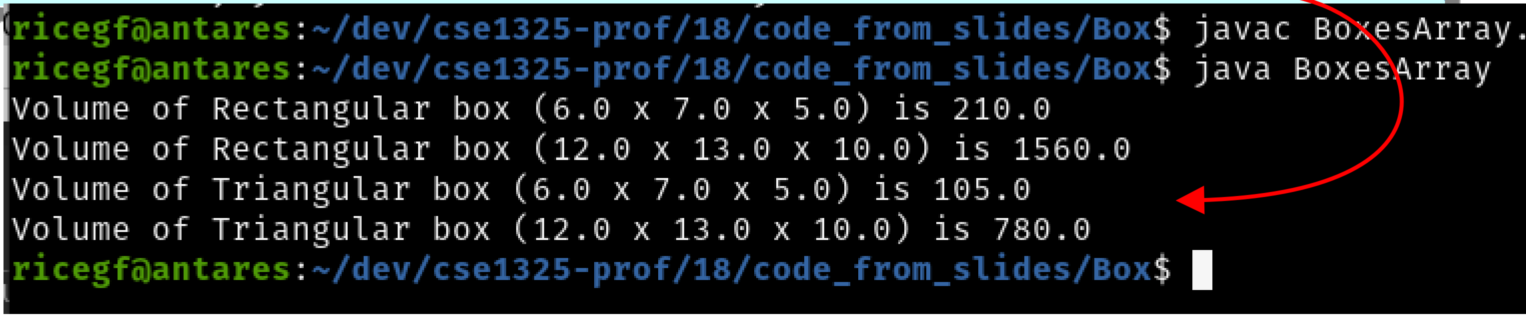
- If an invalid account type is in the file (which I ensured by editing the file!), the user gets this error message

```
#### Error: Bad account type in badtest.moes
java.lang.ClassNotFoundException: customer.Limited
```

Polymorphism

```java
import java.util.ArrayList;

public class BoxesArray {
    public static void main(String[] args) {
        ArrayList<Box> boxes = new ArrayList<>() {{
            add(new Box( 6,  7,  5));
            add(new Box(12, 13, 10));
            add(new TriBox( 6,  7,  5));
            add(new TriBox(12, 13, 10));
        }};

        for(Box box : boxes)
            System.out.println("Volume of " + box + " is " + box.volume());
    }
}
```

Referencing a subclass object from a superclass variable results in calling the *object*'s overridden method, NOT the variable type's method!



```
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ javac BoxesArray.
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ java BoxesArray
Volume of Rectangular box (6.0 x 7.0 x 5.0) is 210.0
Volume of Rectangular box (12.0 x 13.0 x 10.0) is 1560.0
Volume of Triangular box (6.0 x 7.0 x 5.0) is 105.0
Volume of Triangular box (12.0 x 13.0 x 10.0) is 780.0
ricegf@antares:~/dev/cse1325-prof/18/code_from_slides/Box$ 
```

Still no problem!

We may use a subclass object in virtually any place we could use a superclass object. The initialization is another example of upcasting. Upcasting enables **polymorphism**.

Overriding equals and hashCode

- Let’s override equals and hashCode
  - Now our “identity” depends on the field

```
class SSN {
    public SSN(String social) {this.social = social;}
    @Override
    public boolean equals(Object o) {
        if(this == o) return true;         // 1. Is it me?
        if(o == null) return false;        // 2. Is it my type?
        if(this.getClass() != o.getClass()) return false;
        SSN ssn = (SSN) o;                 // 3. Downcast to my type!
        return social.equals(ssn.social);  // 4. Compare significant fields
    }
    @Override
    public int hashCode() {
        return Objects.hash(social);       // List SAME FIELDS as 4. above!
    }
    private String social;
}
```

Upcasting / Downcasting

- Storing a subclass object in superclass variable is called "**upcasting**"
  - No explicit cast syntax is required

```
TriBox t = new TriBox(3,4,5);
Box b = t; // works just fine
```

- Storing a subclass object referenced by a superclass (or interface) variable in a subclass variable is called "**downcasting**"
  - C-like casting is required

```
Box b = new TriBox(3,4,5);
Tribox t = (TriBox) b; // Explicit cast is required
```

  - If the superclass variable is not referencing an object of the subclass type, an exception will be thrown
  - The instanceof operator is helpful to verify types at runtime

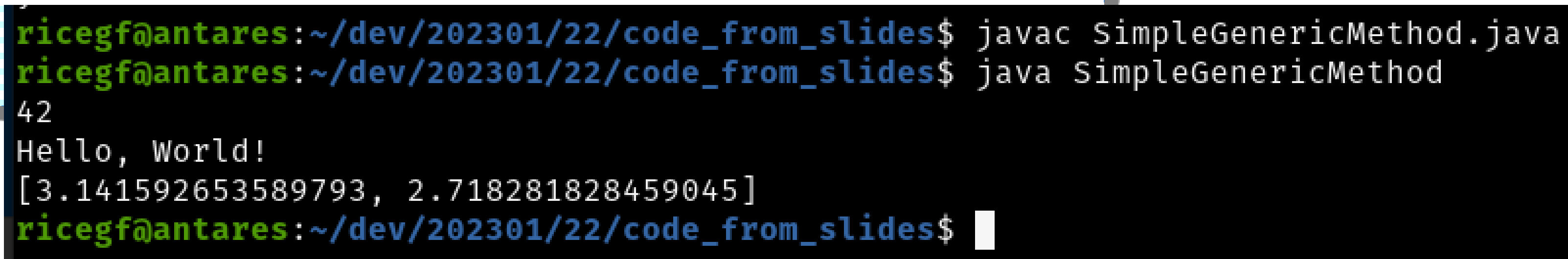
```
Box b = new TriBox(3,4,5);
Tribox t;
if(b instanceof Tribox) t = (TriBox) b;
else t = null;
```

Unconstrained Generic Method

- Write generic methods with <T> before the return type
  - We can then use T (or any other capital letter)
    as a placeholder for the type to be supplied later
  - In Java methods, T’s type is inferred when called

T is the type that is specified when instanced.
<T> before the return type tells Java
our method is generic!

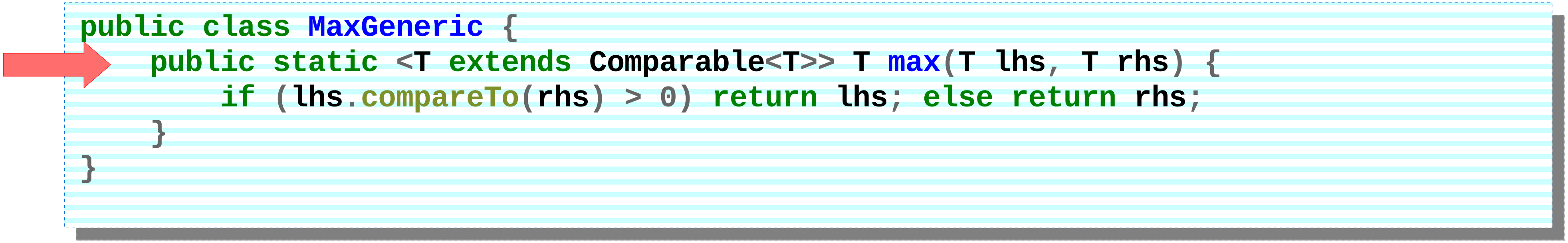
```
import java.util.ArrayList;

public class SimpleGenericMethod {
    public static <T> void printIt(T value) {
        System.out.println(value);
    }
    public static void main(String[] args) {
        printIt(42);
        printIt("Hello, World!");
        ArrayList<Double> doubles = new ArrayList<>();
        doubles.add(Math.PI); doubles.add(Math.E);
        printIt(doubles);
    }
}
```



```
ricegf@antares:~/dev/202301/22/code_from_slides$ javac SimpleGenericMethod.java
ricegf@antares:~/dev/202301/22/code_from_slides$ java SimpleGenericMethod
42
Hello, World!
[3.141592653589793, 2.718281828459045]
ricegf@antares:~/dev/202301/22/code_from_slides$
```

Constrained Generic Method

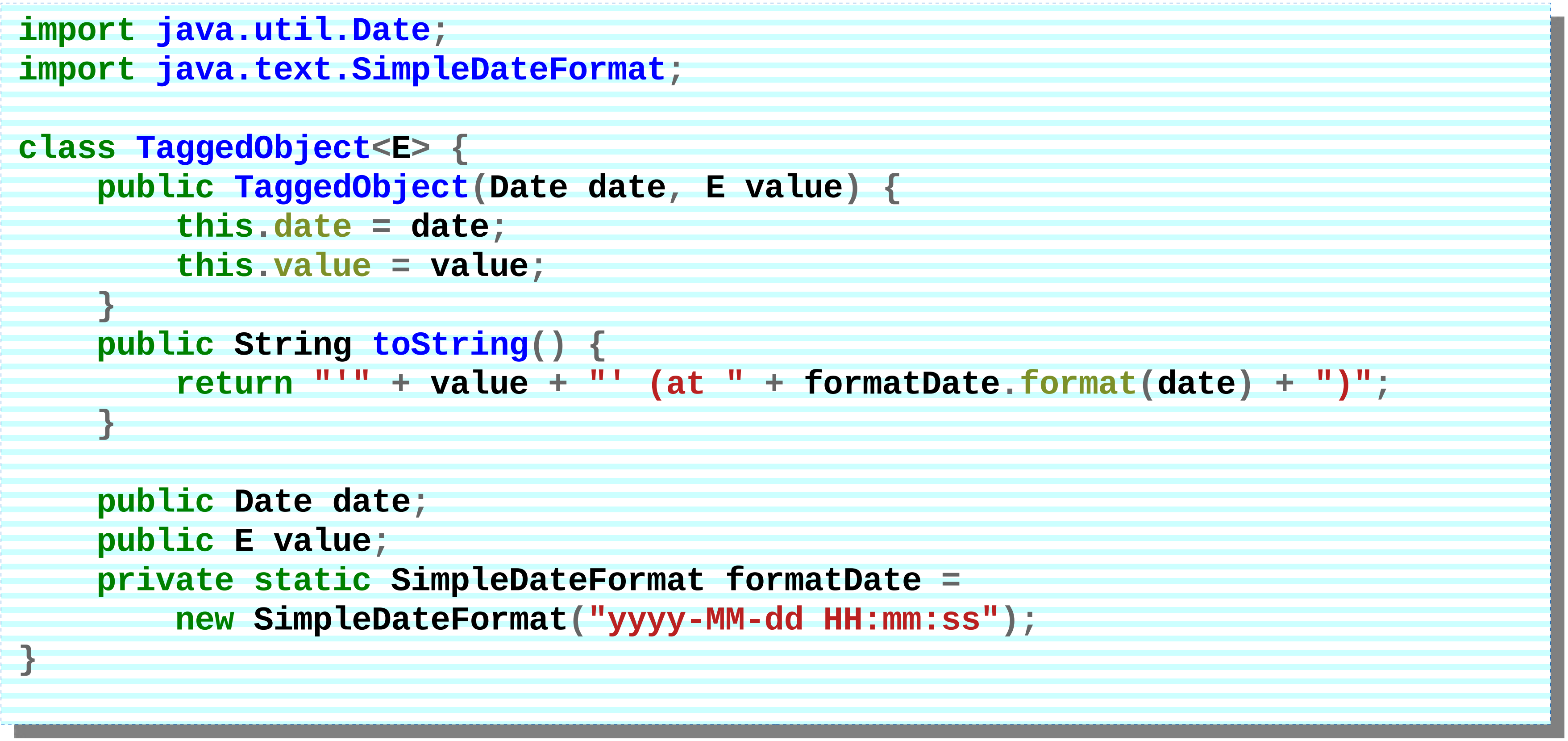
- Instead of a simple **`<T>`** before the return type
  - "Use any non-primitive type that you want"
- We use **`<T extends Comparable<T>>`**
  - "Type T must implement generic interface Comparable for the same type T"



```
public class MaxGeneric {
    public static <T extends Comparable<T>> T max(T lhs, T rhs) {
        if (lhs.compareTo(rhs) > 0) return lhs; else return rhs;
    }
}
```

Generic Class

- A simple <E> after the class name makes it generic



```
import java.util.Date;
import java.text.SimpleDateFormat;

class TaggedObject<E> {
    public TaggedObject(Date date, E value) {
        this.date = date;
        this.value = value;
    }
    public String toString() {
        return "'" + value + "' (at " + formatDate.format(date) + ")";
    }

    public Date date;
    public E value;
    private static SimpleDateFormat formatDate =
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
}
```

Generic Subclass

- Create a generic subclass from a generic class and / or implement generic interfaces using (usually) the same generic variable for each

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

import java.util.Date;
import java.text.SimpleDateFormat;

class TaggedArrayList<E> extends ArrayList<E> implements Comparable<E> {
    // Shadow all of ArrayList's known constructors
    public TaggedArrayList() {
        super();  // Note that super() must ALWAYS be first
        dates = new ArrayList<>();
    }
    public TaggedArrayList(int initialCapacity) {
        super(initialCapacity);
        dates = new ArrayList<>(initialCapacity);
    }
    public TaggedArrayList(Collection<? extends E> c) {
        super(c);
        dates = new ArrayList<>();
        for(E e : this) dates.add(new Date());
    }
```

Inherit from generic superclass and implement generic interface

Continued in original slide deck...

Non-Generic Class from Generic Superclass

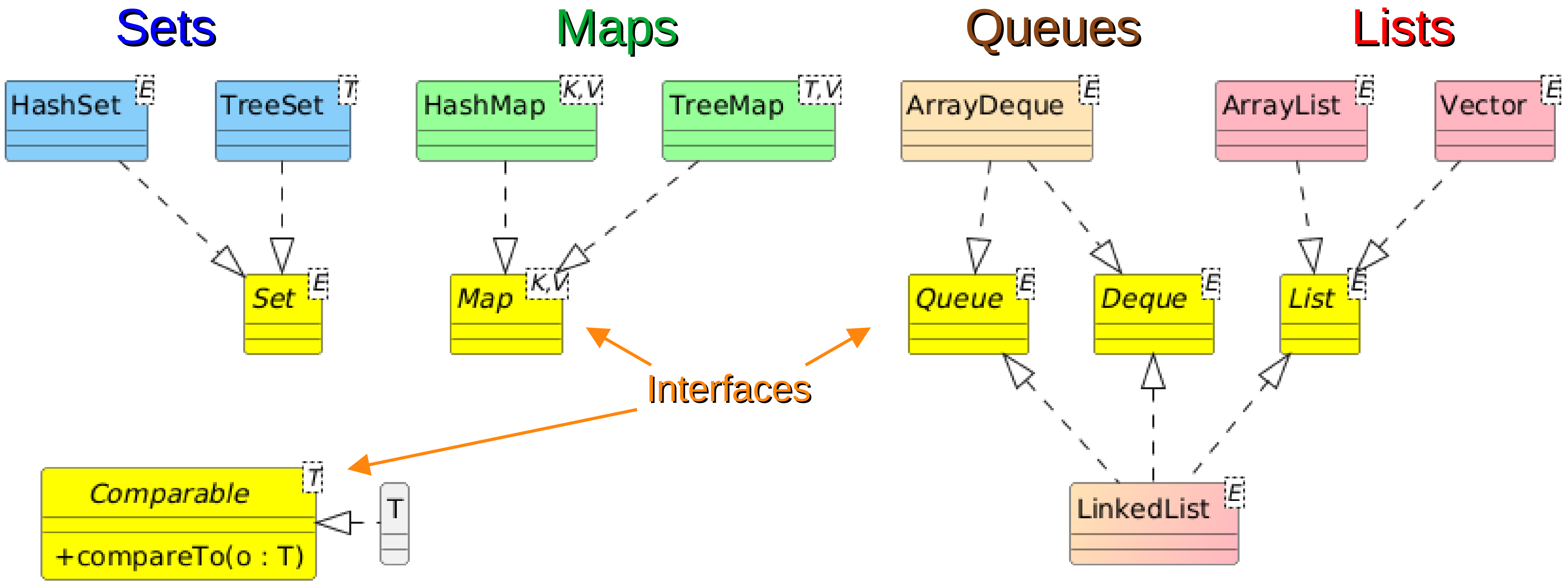
- Create a non-generic subclass from a generic class or interface by specifying a type for the generic superclass or interface

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

import java.util.Date;
import java.text.SimpleDateFormat;

class TaggedIntArrayList extends ArrayList<Integer> {
    // Shadow all of ArrayList's known constructors
    public TaggedArrayList() {
        super();  // Note that super() must ALWAYS be first
        dates = new ArrayList<>();
    }
    public TaggedArrayList(int initialCapacity) {
        super(initialCapacity);
        dates = new ArrayList<>(initialCapacity);
    }
    public TaggedArrayList(Collection<? extends Integer> c) {
        super(c);
        dates = new ArrayList<>();
        for(Integer i : this) dates.add(new Date());
    }
```

Constructors (delegate to ArrayList’s)

Adapted from original slide deck...

JCL Collection Interfaces and their Implementing Classes



Sets

Maps

Queues

Lists

Interfaces

For our sorted
Set and Map

Queue AND List

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
- **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

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

```java
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()

HashSet & TreeSet

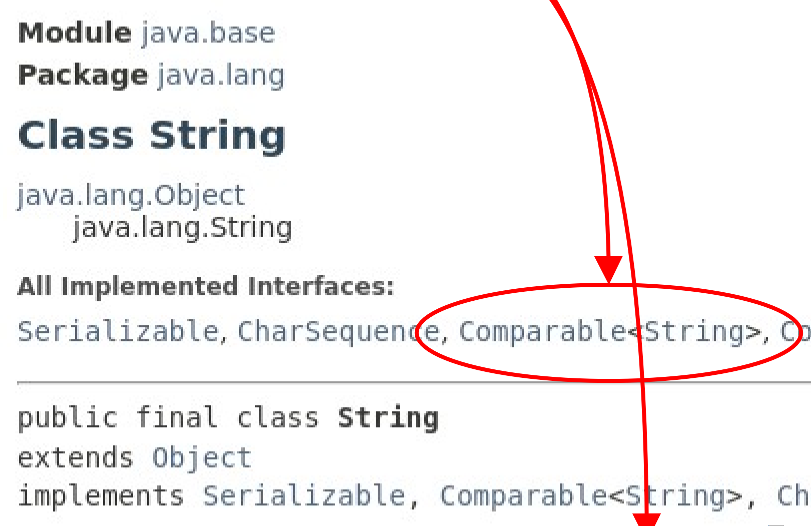
- **HashSet** is a collection of unsorted keys, while **TreeSet** is a collection of sorted keys
  - Essentially an ArrayList of objects with **duplicates automatically removed**, and (for TreeSet) always **sorted**
  - Objects stored in HashSet / TreeSet MUST override **`hashCode`**!
  - For TreeSet, 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

```
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

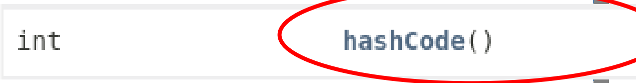
Class String

java.lang.Object
    java.lang.String

All Implemented Interfaces:

Serializable, CharSequence, Comparable<String>, Co

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



int	hashCode()

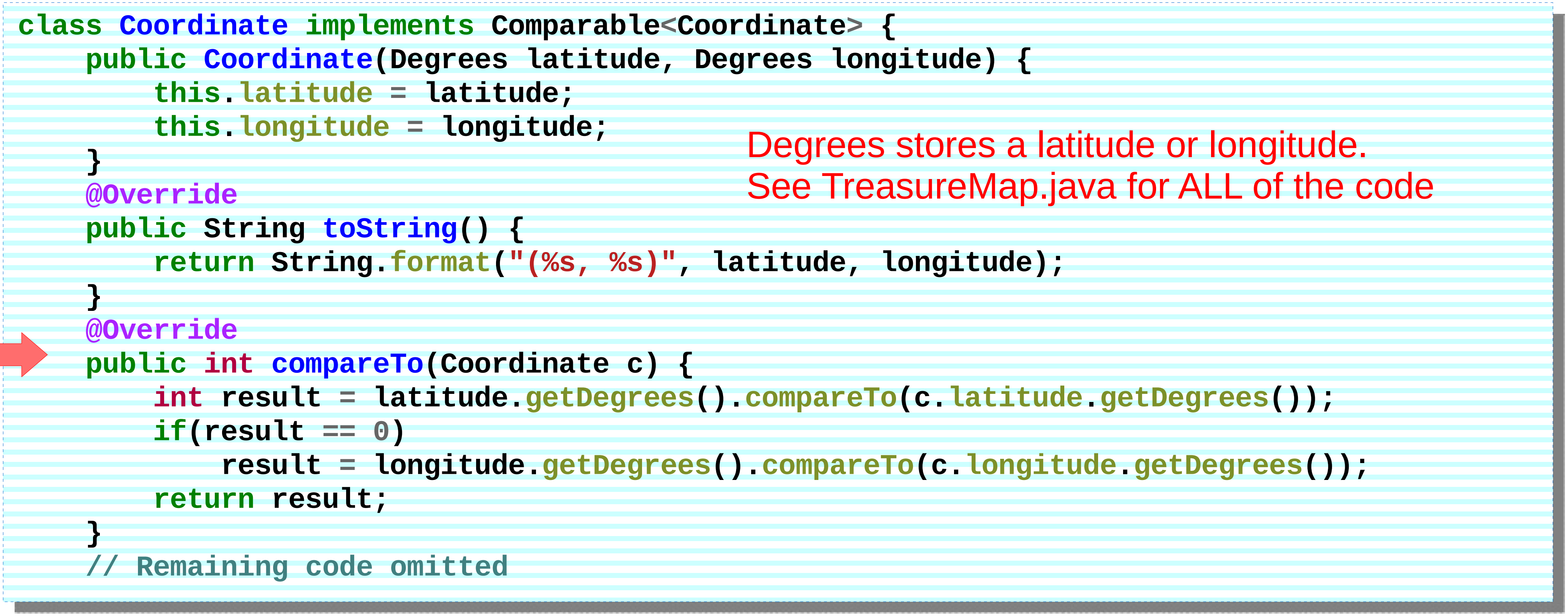
HashMap & TreeMap

- **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

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

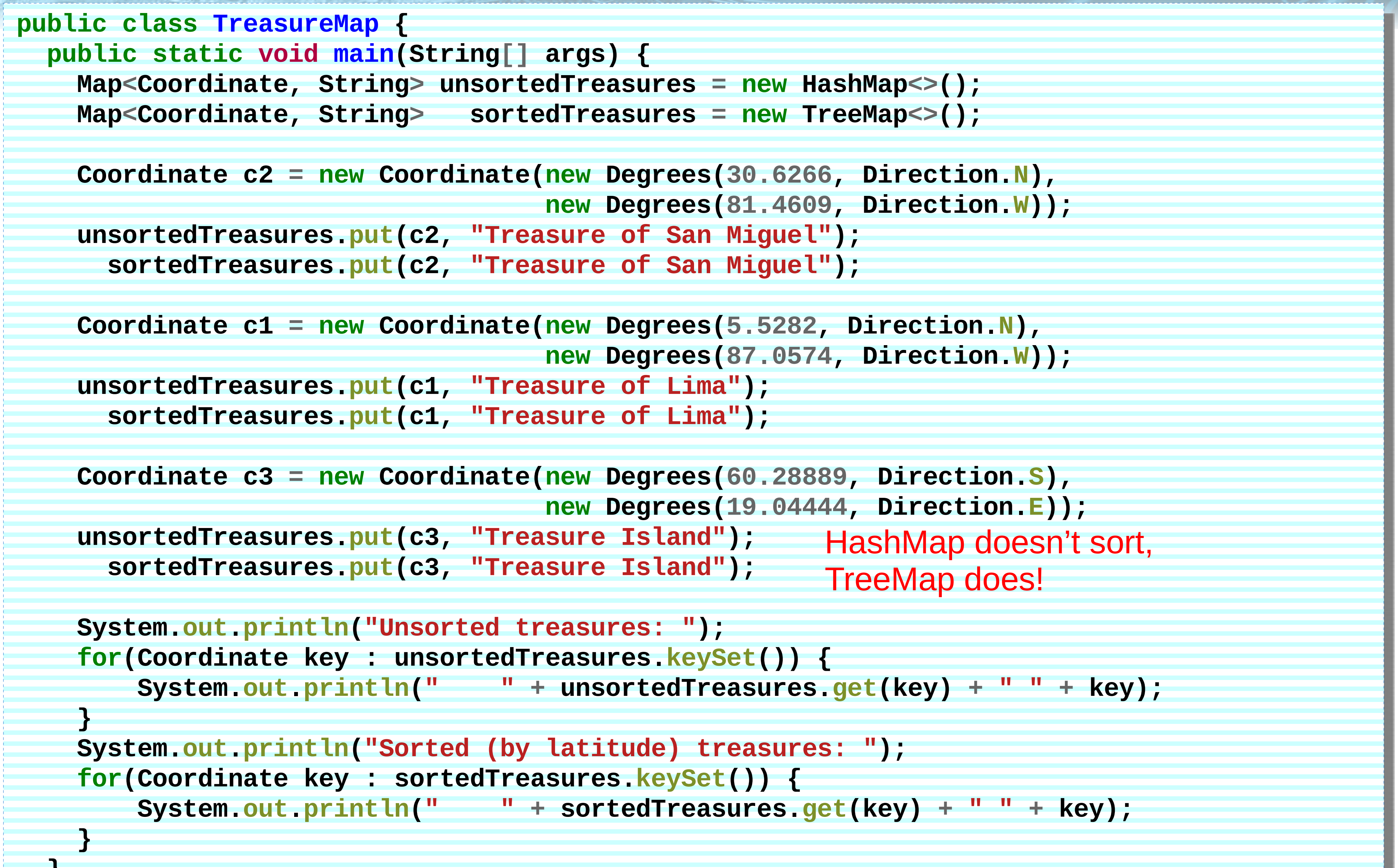
- Class coordinate stores lat-longs around the globe
- BEWARE: If your compareTo method returns 0 (“equals”) then the new element will *overwrite* an existing Map entry!
  - That’s why compareTo also checks longitude



```
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



```java
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!

equals & hashCode example

```
class Treasure {
    public Treasure(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;
        Treasure t = (Treasure) o; // Downcast to a Treasure
        return coordinate.equals(t.coordinate)      // class type
            && treasureName.equals(t.treasureName)  // String type
            && (treasureValue == t.treasureValue);  // primitive type
    }
    @Override
    public int hashCode() {
        return Objects.hash(coordinate, treasureName, treasureValue);
    }
    private Coordinate coordinate;  // Our custom class (Roving Robots, enhanced)
    private String treasureName;    // A JCL class
    private double treasureValue;   // A primitive
}
```

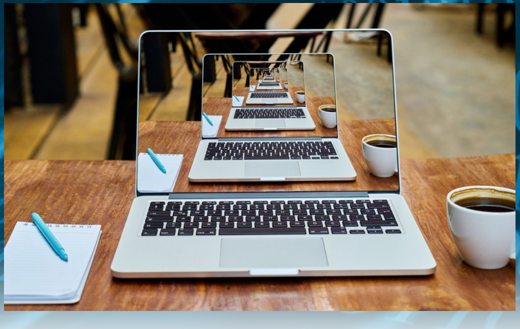
For this example class, assume Coordinate has already defined equals() and hashCode()

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);`

Iterator (Interface)



- **Iterator**: A pointer-like object used to access items managed by a Collection
- Iterator (from collection’s **iterator()** method) 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()
- ListIterator (from collection’s **listIterator()** method) subclasses Iterator, adding 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/Iterator.html
https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/ListIterator.html

A ListIterator Example

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

```
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());
}
```

Move tomato to fruit!

Thread Pools (Concurrency) with a Synchronized Mutex

```
public void calculateImageViaPool (int numThreads) {
    Thread[] threads = new Thread[numThreads];
    for(int i=0; i<numThreads; ++i) {
        threads[i] = new Thread(() -> calculateRows());
        threads[i].start();
    }
    for(Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted Exception");
        }
    }
}
private static Object mutex = new Object();
private void calculateRows() {
    int row = 0;
    while(true) {
        synchronized(mutex) {
            row = nextY++;
        }
        if(row >= height) break;
        calculateRow(row);
    }
}
```

Instance new threads with a lambda
(and don’t forget to start them!)

Join a thread to wait for it to complete

Synchronize on a static Object
to avoid thread interference

Thread Pools
with a Synchronized Method

```
public void calculateImageViaPool (int numThreads) {
    Thread[] threads = new Thread[numThreads];
    for(int i=0; i<numThreads; ++i) {
        threads[i] = new Thread(() -> calculateRows());
        threads[i].start();
    }
    for(Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted Exception");
        }
    }
}
private synchronized int nextRow() {
    return nextY++;
}
private void calculateRows() {
    int row = 0;
    while(true) {
        row = nextRow();
        if(row >= height) break;
        calculateRow(row);
    }
}
```

Create Thread[ ] or ArrayList<Thread>
Loop: Instance Thread class using lambda
Start the thread
Loop: Join the threads to wait for completion

Or synchronize a method
to avoid thread interference

Don’t Forget!

- **Vocabulary!** The exact same definitions on the study sheet will be on the exam (except parentheticals – word within parentheses)
- **Concepts!** We’ll ask 15 multiple-choice questions, a few from each lecture
- **Coding!** We’ll ask you to code small sections of 2 or 3 larger applications, each demonstrating a few of the techniques we’ve practiced
- **Comprehensive!** You’ll need to know the basics of what we covered earlier this semester to understand and answer the new questions

For Next Class

- **Exam #2** (16⅔% of final grade)
- Study sheet and practice exams are on Canvas

Questions?

Owl by PeterKraayvanger per the Pixabay License
https://pixabay.com/photos/owl-bird-animal-uhu-european-uhu-62703/