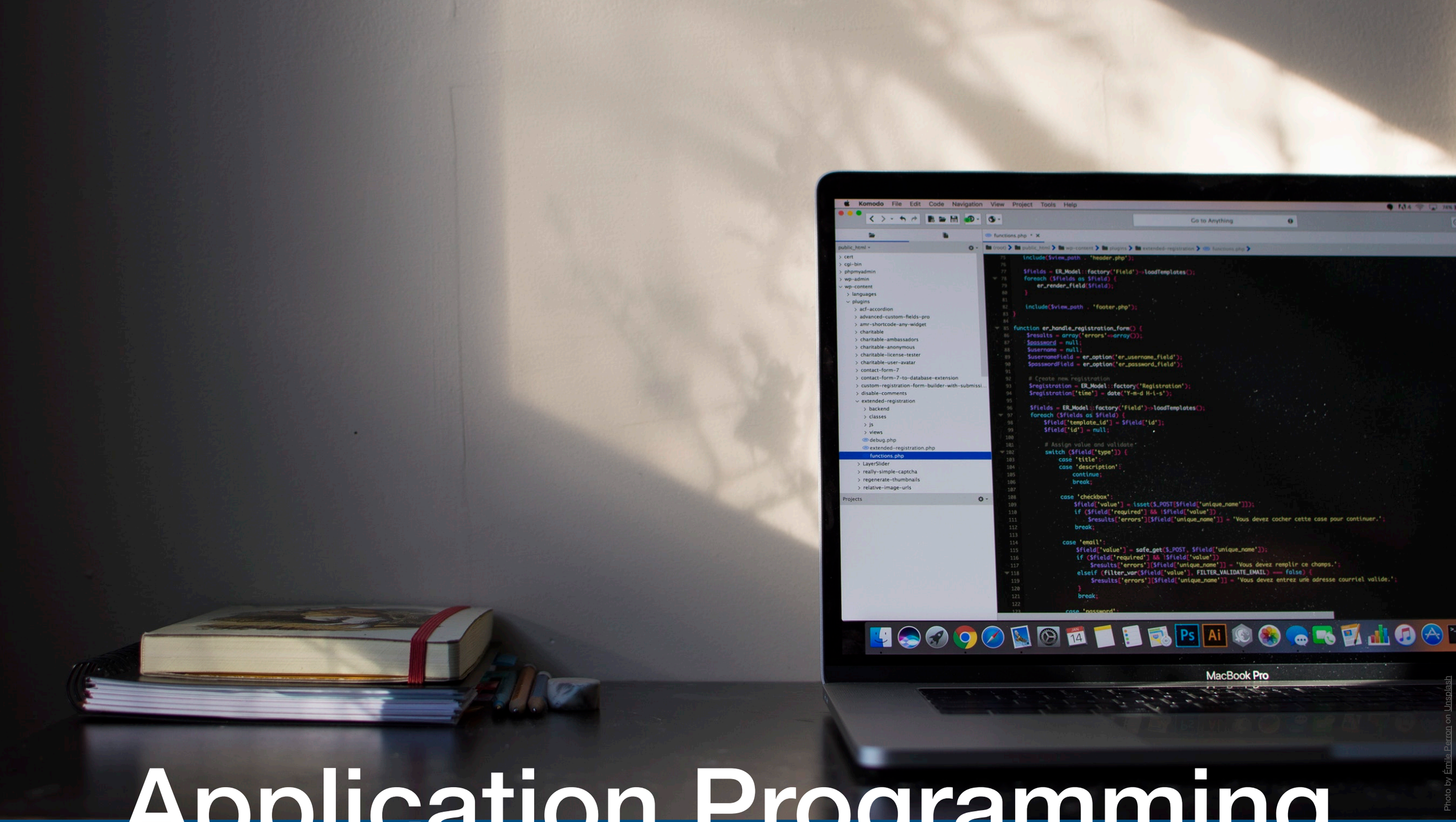


Application Programming

UTSA Computer Science 3443

Fall 2022

Dr. Amanda Fernandez

Week 11

- Collections
- Generics

- Lab 5 due Saturday
- Quiz 5 due Saturday

Fall 2021*

Week	Dates	Topic	Lab	Quiz	Chapter(s)
1	8/23 - 8/27	Introductions, Syllabus, Java basics, Eclipse			1-6
2	8/30 - 9/3	Java & OOP Concepts, Javadoc	1	1	8,14
3	9/6 - 9/10	ArrayList, Strings			7,15
4	9/13 - 9/17	File I/O, UML, MVC	2	2	12,13
5	9/20 - 9/24	Introduction to JavaFX			
6	9/27 - 10/1	JavaFX, SOLID	3	3	
7	10/4 - 10/8	Introduction to Git, *Review*			
8	10/11 - 10/15	**Midterm Exam**			
9	10/18 - 10/22	JavaFX applications	4	4	13,20
10	10/25 - 10/29	Exception Handling			11
11	11/1 - 11/5	Collections & Generics			16, 19
12	11/8 - 11/12	JUnit Testing, logging, Application Design	**5**	**5**	
13	11/15 - 11/19	Concurrency & Multithreading + ***Review***			17
14	11/22 - 11/24*	Lambda Expressions (+*Thanksgiving*)	**6***	**6***	21
15	11/29 - 12/3	**Team project demos**			
-	12/6 - 12/10	**Final Exams**			

Generics

Generics

- In a nutshell, **generics** enable *types* (classes and interfaces) to be parameters when defining classes, interfaces and methods.

- For example, let's create a list of Strings without generics..
  - Declare the list, in this case as an ArrayList.

```
List list = new ArrayList();
```

  - Add a String object to the list.

```
list.add("hello");
```

  - In order to retrieve it from the list, we must **cast** it to a String type.

```
String s = (String) list.get(0);
```

Generics

- Instead, we can leverage **generics** to eliminate the need for casting in this example.
- Same example, let's create a list of Strings.
  - Declare the list, in this case as an ArrayList.

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

  - Add a String object to the list.

```
list.add("hello");
```

  - In order to retrieve it from the list, no need to cast!

```
String s = list.get(0);        // no cast needed!
```

Generics

- Advantages to using generics in your code:
    - Eliminates casting
        - *As in previous example.*
    - Stronger type checks are performed at compile time
        - If a problem exists, it's better to find it at compile time than at run time!
    - Enables programmers to implement generic algorithms
        - Focus is on creating more elegant algorithms, rather than syntax.
        - Still generates clean, type-safe, customizable code.

Generic Methods

- Creating generic methods enables code reuse, simplifying your code.

- For example, let's write a method which:

    - Takes in an array of numbers, and
    - Returns a random element in the array.

```
public Integer randomInteger(Integer[] array) {
      Random random = new Random();
      int index = random.nextInt(array.length);
      Integer result = array[index];
      return result;
 }
```

Generic Methods

- Another example?
- Let's write a method which:
    - Takes in an array of Strings, and
    - Returns a random element in the array.

```
public String randomString(String[] array) {
      Random random = new Random();
      int index = random.nextInt(array.length);
      String result = array[index];
      return result;
 }
```

Generic Methods

- Both examples require the same code!

```
public Integer randomInteger(Integer[] array) {
      Random random = new Random();
      int index = random.nextInt(array.length);
      Integer result = array[index];
      return result;
  }
```

```
public String randomString(String[] array) {
      Random random = new Random();
      int index = random.nextInt(array.length);
      String result = array[index];
      return result;
  }
```

Generic Methods

- Instead, let's write a generic method for this….

```
public <T> T randomElement(T[] array) {
        Random random = new Random();
        int index = random.nextInt(array.length);
        T result = array[index];
        return result;
    }
```

<T>	indicates that the method will be a generic method
T	indicates the type of the generic variable in the method *(defined when the method is called!)*

Generic Methods

- Instead, let's write a generic method for this….

```
public <T> T randomElement(T[] array) {
        Random random = new Random();
        int index = random.nextInt(array.length);
        T result = array[index];
        return result;
    }
```

- Now we can call this one method with either..

```
Integer[] intArray = { 1, 3, 5, 7, 9, 0, 2, 4, 6, 8 };
String[] stringArray = { "xx", "yy", "zz", "aa", "bb", "cc" };

Integer rint = this.randomElement(intArray);
String rstring = this.randomElement(stringArray);
```

Generic Classes

- A ***generic class*** is a class that is parameterized over types.

- One of the key concepts of Java's generics is that *only the compiler* processes the generic parameters.
  - The Java compiler uses generics to ensure type-safety.
  - There is no generic type-checking in the runtime code.

- Essentially, generic classes avoid the same issues faced in our previous example for generic methods.
  - **Generics.zip** *contains PairTest.java and PairOfSameType.java*

Collections

Java Collections

- In Java, `Collection` is an interface defining the behaviors of a collection, including typical operations such as…
  - add elements to the collection, `.add(..)`
  - access elements of the collection, `.get(..)`
    - loop over the elements in the collection, `iterator`
    - access an element by its index, if it applicable (ie: List)
  - test whether an element is contained in the collection, `.contains(..)`
  - find out the size of the collection `.size()`
  - remove elements from the collection. `.remove(..)`

Java Collections History

- JDK 1.0: Vector, Dictionary, Hashtable, Stack, Enumeration
- JDK 1.2: Collection, Iterator, List, Set, Map, ArrayList, HashSet, TreeSet, HashMap, WeakHashMap
- JDK 1.4: RandomAccess, IdentityHashMap, LinkedHashMap, LinkedHashSet
- JDK 1.5: Queue, java.util.concurrent, ...
- JDK 1.6: Deque, ConcurrentSkipListSet/Map, ...
- JDK 1.7: TransferQueue, LinkedTransferQueue

Java Collections

- There are many in Java.. A few of the most common are listed:
  - **`List`**
    - `ArrayList`
    - `LinkedList`
  - **`Set`**
    - `HashSet`
    - `TreeSet`
  - **`Map`**
    - `HashMap`
    - `TreeMap`

`List`, `Set`, and `Map` are all interfaces!

Java Collections

- Common practice: leverage generics when initializing a collection.

- Typically, a collection is declared by including the type of element it contains within `<...>`, which is using Java's *generics* notation.

- For example:

```
// declare list to be a collection of Strings
Collection<String> list;

// initialize list to a concrete class that implements Collection
list = new ArrayList<String>();
```

Collections & Primitive Types

- Java does *not* support collections of primitive types.

- However, each primitive type has a corresponding *wrapper class*, which enables you to manipulate primitive type values as objects. For example:
  - `double` has `Double`
  - `int` has `Integer`

- The conversion between these is mostly automatic..
  - primitive type to wrapper class conversion = **autoboxing**
  - wrapper class object to primitive type conversion = **unboxing**

Autoboxing & Unboxing

- Java makes these conversions mostly transparent..

```
Double dbox = Math.sqrt(2);   // autoboxing

double d = 1.0 / dbox;        // unboxing
```

- But it is important to understand what is happening, because they may result in counterintuitive errors.

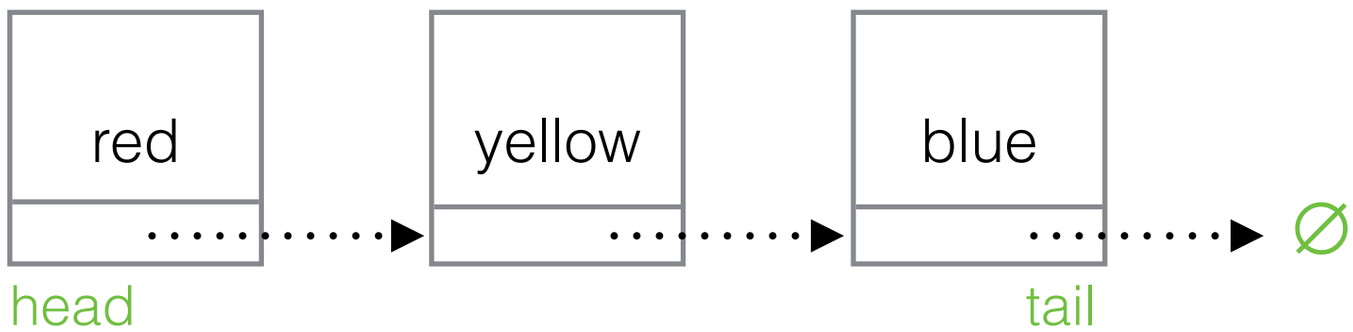
```
Integer i1 = 42;
Double i2 = (double) 42;
System.out.println(i1 == i2); // This line doesn't compile!

// Compile-time error message:
//   "Incompatible operand types Integer and Double."
```

List

- java.util.List is an interface defining an ordered collection (also known as a sequence).

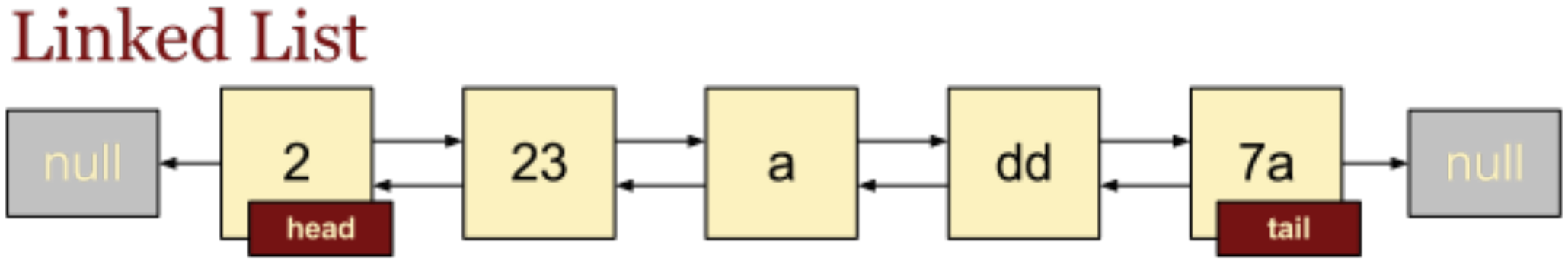
- `LinkedList` implements `List`
  - For example, add “red”, “yellow”, and “blue” to a new LinkedList object.



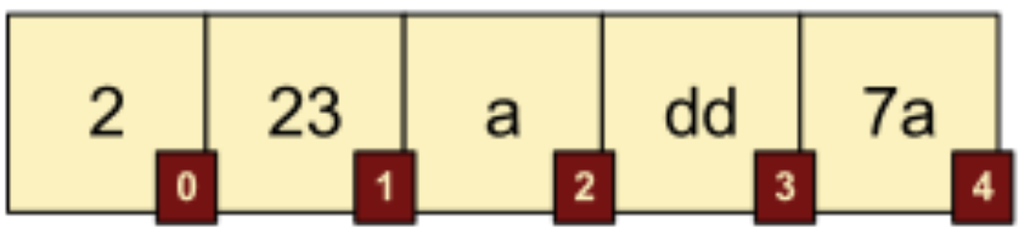

```
List<String> colorList = new LinkedList<String>();
colorList.add(“red”);
colorList.add(“yellow”);
colorList.add(“blue”);
```

LinkedList vs. Array

Linked List



Array



List

- So what's the difference? Why use different types of lists??
    - One reason: performance..

	add	remove	get	contains
ArrayList	O(1)	O(n)	O(1)	O(n)
LinkedList	O(1)	O(1)	O(n)	O(n)
CopyOnWrite ArrayList	O(n)	O(n)	O(1)	O(n)

Set

- java.util.Set is an interface defining a collection which contains no duplicate elements.
    - For example, if I have the following array of colors, and I print them as a list…

```
String[] colors = { “red","white","blue","green","gray",“orange","tan","white",
            "cyan","peach","gray","orange" };


List<String> list = Arrays.asList(colors);
System.out.printf("List: %s%n", list);
```

console:

```
List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]
```

Set

- Now if I save that list to a set object, the contents of that collection will change.
    - Since duplicates are not allowed in sets, the new set object will contain only one of each color.

```
String[] colors = { “red","white","blue","green","gray",“orange","tan","white",
                    "cyan","peach","gray","orange" };

List<String> list = Arrays.asList(colors);
System.out.printf("List: %s%n", list);

Set<String> set = new HashSet<>(list);
System.out.printf("Set: %s%n", set);
```

console:

```
List: [red, white, blue, green, gray, orange, tan, white, cyan, peach, gray, orange]
Set: [tan, green, peach, cyan, red, orange, gray, white, blue]
```

Set

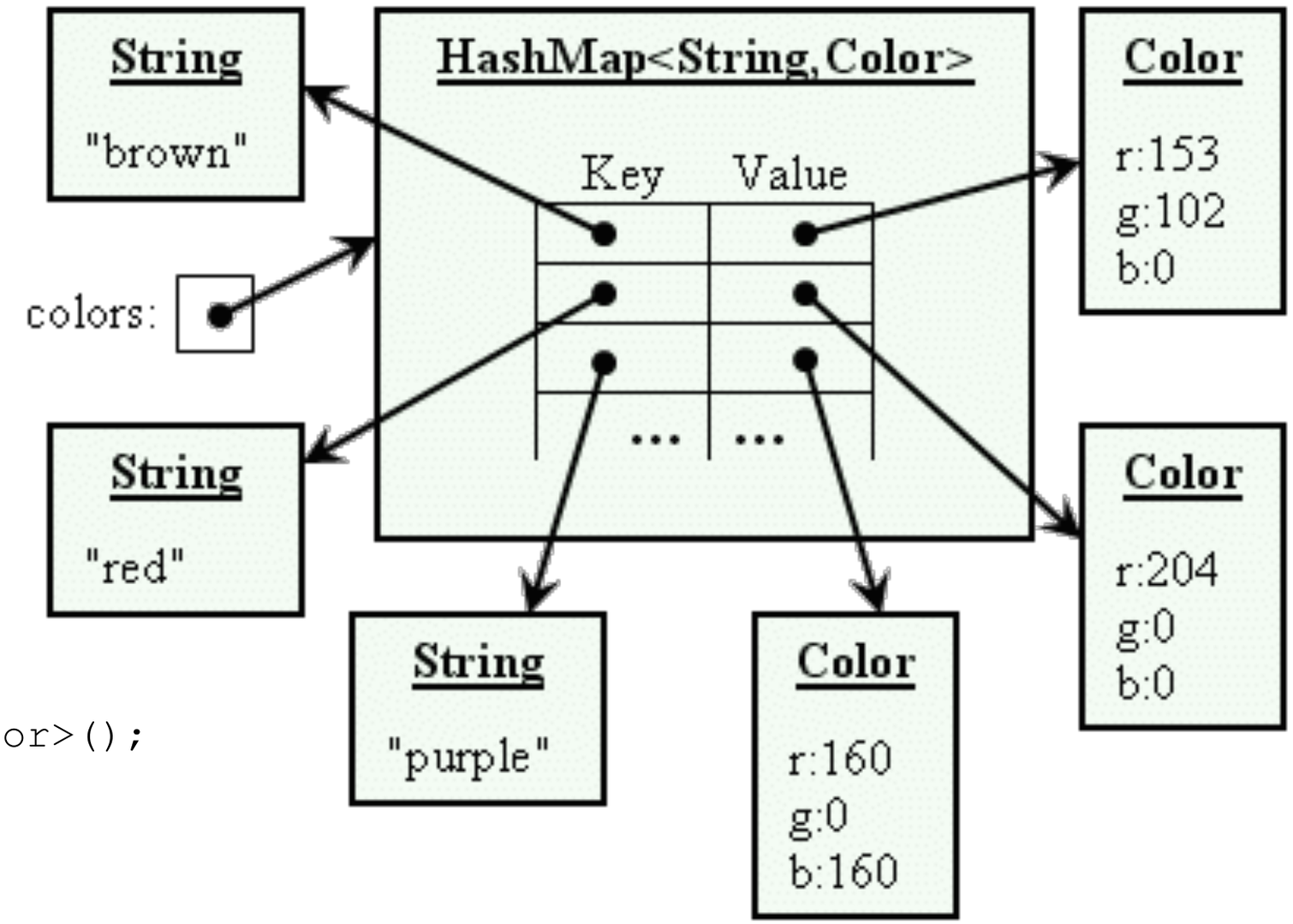
	Data Structure	Sorting	Iterator	Nulls?
HashSet	Hash table	No	Fail-fast	Yes
Linked HashSet	Hash table + linked list	Insertion Order	Fail-fast	Yes
EnumSet	Bit vector	Natural Order	Weakly consistent	No
TreeSet	Red-black tree	Sorted	Fail-fast	Depends
CopyOnWrite ArraySet	Array	No	Snapshot	Yes
Concurrent SkipListSet	Skip list	Sorted	Weakly consistent	No

Set

	add	contains	next
HashSet	O(1)	O(1)	O(h/n)
Linked HashSet	O(1)	O(1)	O(1)
EnumSet	O(1)	O(1)	O(1)
TreeSet	O(log n)	O(log n)	O(log n)
CopyOnWrite ArraySet	O(n)	O(n)	O(1)
Concurrent SkipListSet	O(log n)	O(log n)	O(1)

Map

- java.util.Map is an interface defining a collection which maps keys to values.
    - Keys can map to at most one value.
    - No duplicate keys.




```
Map<String,Color> colors;
colors = new HashMap<String,Color>();
```

Map

- java.util.Map is an interface defining a collection which maps keys to values.
    - For example, if I have the following student names and IDs…

```
// I would have to maintain two arrays, one for names and one for IDs..
String[] studentNames = {“Alice”, “Bob”, “Carlos”, “Diane”};
String[] studentIDs = {“atf123”, “ght456”, “liw789”, “pwt012”};

// Then print out Alice, I need to know she is at index 0..
System.out.println( studentNames[0] + “ “ + studentIDs[0] );

// Instead, let’s use a map!
Map<String,String> classMap = new HashMap<String,String>();
classMap.put(“atf123", "Alice");              // As students register for the class,
classMap.put("ght456", "Bob");                //    I can add them to the map. Then to
classMap.put("liw789", "Carlos");             //    retrieve them, I only need their ID.
classMap.put("pwt012", "Diane");
System.out.println( classMap );
```

Map

- java.util.Map is an interface defining a collection which maps keys to values

	Data Structure	Sorting	Iterator	Nulls?
HashMap	Hash table	No	Fail-fast	Yes
LinkedHashMap	Hash table + linked list	Insertion or access order	Fail-fast	Yes
IdentityHashMap	Array	No	Fail-fast	Yes
WeakHashMap	Hash table	No	Fail-fast	Yes
EnumMap	Array	Natural order	Weakly consistent	No
TreeMap	Red-black tree	Sorted	Fail-fast	Yes
ConcurrentHashMap	Hash tables	No	Weakly consistent	No
ConcurrentSkipListMap	Skip list	Sorted	Fail-fast	No

MAP

- Different maps have different functionality, and different performance..

	get	containsKey	next
HashMap	O(1)	O(1)	O(h/n)
LinkedHashMap	O(1)	O(1)	O(1)
IdentityHashMap	O(1)	O(1)	O(h/n)
WeakHashMap	O(1)	O(1)	O(h/n)
EnumMap	O(1)	O(1)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)
ConcurrentHashMap	O(1)	O(1)	O(h/n)
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)

Map

- Example: phone book

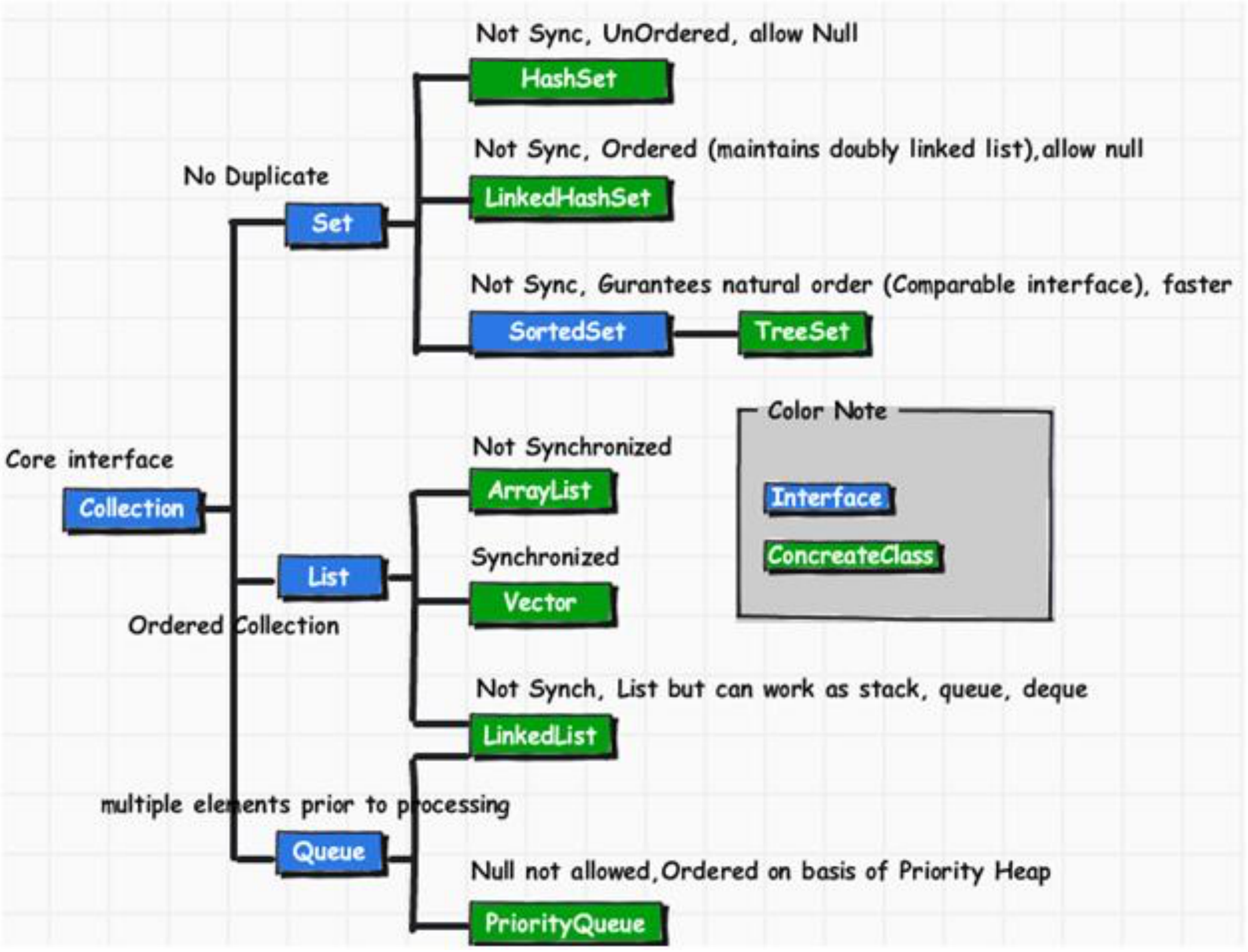
```
// Phone book implementation
Map<String,PhoneNumber> phoneBook = new HashMap<String,PhoneNumber>();
phoneBook.put(“Alice", new PhoneNumber(“210-555-1234”));
phoneBook.put("Bob", new PhoneNumber(“210-555-4321”));
phoneBook.put("Carlos", new PhoneNumber(“210-555-4444”));
phoneBook.put("Diane", new PhoneNumber(“210-555-1111”));

System.out.println( phoneBook );
```

```
// Class to handle phone numbers
public class PhoneNumber{
    private String number;

     public PhoneNumber( String phoneNumber ){
        this.number = phoneNumber;
     }
}
```

JAVA COLLECTIONS



Class Exercise

- *What collection types would you use in the following examples?*

1. A phone book (name, phone number)
2. Storing user interaction history (clicks, actions, choices, etc)
3. An address book (name, phone number, address, etc)
4. User choices for character attributes in a game (hair color, shoes, etc).
5. Ordered task manager.

Activity

- Come up with 3 distinct applications:

1. Requires a List
2. Requires a Set
3. Requires a Map

```
// TODO: Need a map with state names and a list of cities
        //   key = state name
        // value = listing of cities in that state

        Map<String,List<String>> states = new HashMap<String,ArrayList<String>>();

        List<String> tx = new ArrayList<String>();
        tx.add( "San Antonio" );
        tx.addAll( Arrays.asList("Austin", "Dallas", "Corpus Christi", "Dallas") );
        states.put("Texas", tx );

        List<String> ny = new ArrayList<String>();
        ny.addAll( Arrays.asList("NYC", "Albany", "Niagara", "Long Island") );
        states.put("New York", ny );

        System.out.println( states );
```

In-lecture examples

- **EBookReader** example - is an ArrayList still a good choice for our library of books?
    - …Maybe not?
    - We could update the Model to maintain the collection of books as a HashMap.

JavaFX + Collections

JavaFX + Collections

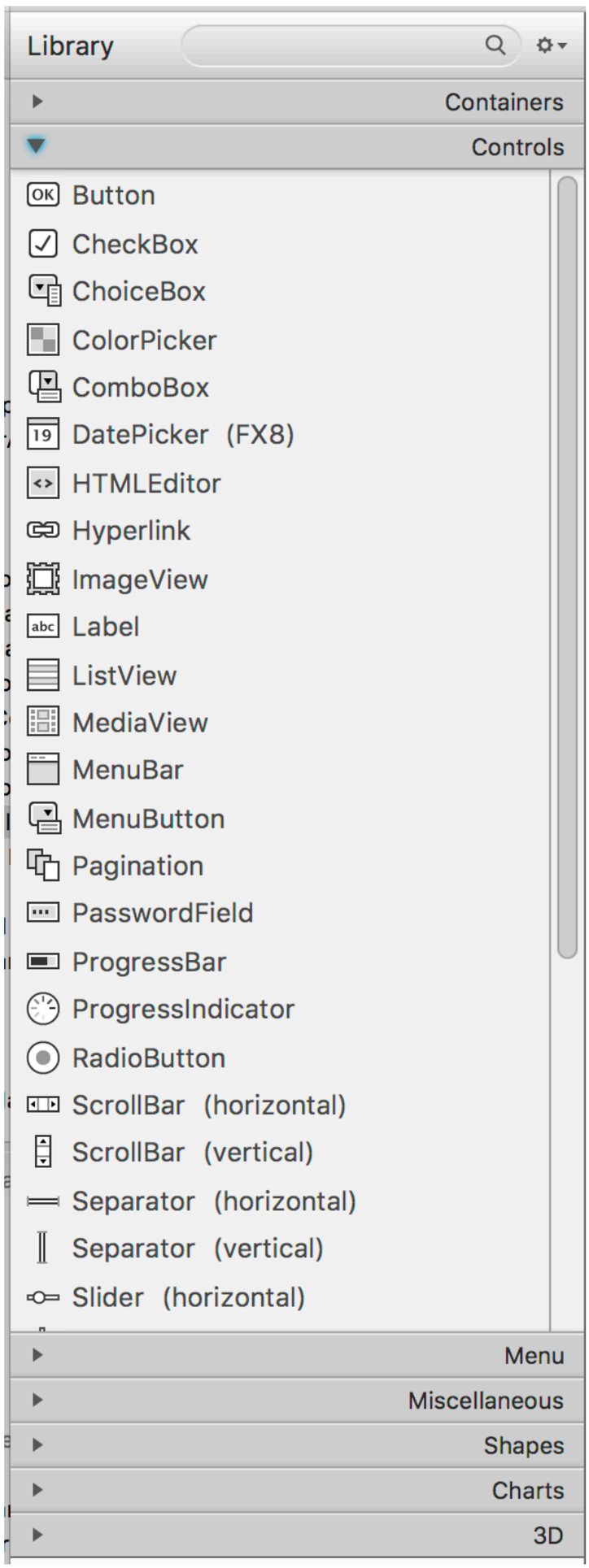
- Let’s try a new GUI component: **ListView**
- To integrate a list collection with this component, we should translate it to an **ObservableList**

https://docs.oracle.com/javase/8/javafx/api/javafx/collections/ObservableList.html

JavaFX

- JavaFX provides several GUI *controls.*
- Let's try **ListView** for displaying data.

1. FXML: Add a ListView using SceneBuilder
2. Controller:
    1. Implement the Initializable interface
    2. Connect ListView to controller
    3. Create an **ObservableList**
    4. Add your data to this list
    5. Connect the list to the ListView



ListView

```
public class MainController implements EventHandler<ActionEvent>, Initializable {

    @FXML
    ListView<String> dispList;
    ObservableList<String> items = FXCollections.observableArrayList();

    @Override
    public void handle(ActionEvent event) {
        dispList.getItems().add( “your name here!” );
    }

    @Override
    public void initialize(URL location, ResourceBundle resources) {
        ArrayList<String> names = new ArrayList<String>();
        names.add( "Alice" );
        names.add( "Bob" );
        names.add( "Carlos" );

        items.addAll( names );
        dispList.getItems().addAll(items);
    }

}
```

PieChart

```
public class MainController implements EventHandler<ActionEvent>, Initializable{

    @FXML
    PieChart pieChart;


    @Override
    public void initialize(URL location, ResourceBundle resources) {

        ArrayList<PieChart.Data> expenses = new ArrayList<PieChart.Data>();
        expenses.add( new PieChart.Data( "food", 101.37 ) );
        expenses.add( new PieChart.Data( "transportation", 11.41 ));
        expenses.add( new PieChart.Data( "coffee", 2.10 ));
        expenses.add( new PieChart.Data( "textbook", 37.03 ));

        ObservableList<PieChart.Data> data = FXCollections.observableList( expenses );

        pieChart.setData( data );

    }
}
```