1 **Generic Collections**

CST242

2 **Generic Classes (Page 1)**

- A collection is an object that represents a group of elements
- A generic class is any collection class which specifies an additional *generic type* that limits the type elements may be stored in the collection
  – E.g. ArrayList<Double> (an ArrayList of Doubles) or LinkedList<String> (a LinkedList of Strings)

3 **Generic Classes (Page 2)**

- Formats to declare:
  CollectionType<Type> *object* = new *Constructor*<Type>();
  CollectionType<Type> *object* = new *Constructor*<>();
  CollectionType<Type> *object* = new *Constructor*();
  – It is considered *redundant* by the Java compiler to recode *Type* (the generic type) within the *Constructor* call or even to include the <chevrons>

4 **Generic Classes (Page 3)**

- Examples:
  ArrayList<String> names = new ArrayList<String>();
  ArrayList<String> names = new ArrayList<>();
  ArrayList<String> names = new ArrayList();

5 **Wrapper Classes (Page 1)**

- Wrapper classes include Boolean, Character, Double, Float, Byte, Short, Integer and Long
- Recall that the methods for parsing Strings to numerics are members of wrapper classes, e.g.
  Double.parseDouble
- These classes allow any primitive variable to be treated as an object

6 **Wrapper Classes (Page 2)**

- Collection classes and other data structures are not able to manipulate primitives so they must use the wrapper classes in their declarations
  ArrayList<Double> numbers = new ArrayList();
- Wrapper classes are located in the java.lang package and do *not* need to be imported

7 **Autoboxing and Auto-Unboxing (Page 1)**

- Java automatically convert between primitive-type values and wrapper-type objects
- Autoboxing is the process of converting from a primitive-type value to a wrapper-type object
- Auto-Unboxing is the process of converting from a wrapper-type object to a primitive-type value

- Boxing and unboxing occur often when storing and retrieving data from generic collections

8 **Autoboxing and Auto-Unboxing (Page 2)**

- Example:
  Integer valueObject = 10; // Boxing
  int value = valueObject; // Unboxing

9 **The Collection Interface**

- The Collection interface is the foundation upon which the collections framework is built
- It declares methods that all collections will have which perform operations on the *entire* collection
- Interfaces List, Set and Queue all are derived from Collection
- Found in the java.util package
  import java.util.Collection;

10 **The List Interface (Page 1)**

- A List is a Collection of elements in sequence that may include duplicates and is *index* accessed (zero-based)
- Classes that *implement* List include ArrayList and LinkedList
- Use one of the List types in favor of an array if the collection needs to be dynamically resized
- Found in the java.util package
  import java.util.List;

11 **The List Interface (Page 2)**

- Format to instantiate a List from an ArrayList:
  List<Type> *object* = new ArrayList();
  – *Type* is the generic type
- Example:
  List<Double> list = new ArrayList();

12 **Subtyping**

- Types defined by a subclass definition actually are subtypes of their superclass
- If the ArrayList interface is an extension of the interface List:
  – The interface object:
    List<Double> list;
  – Can be instantiated by calling its subtype constructor:
    list = new ArrayList();
  – Or in a single "declare and instantiate" statement:
    List<Double> list = new ArrayList();

13 **The add() Method**

- The add() method of the List interface appends the specified *element* to the end of

the list
- Format:
  listObject.add(*element*)
- Example:
  list.add(number);

15 **The Iterator Interface (Page 1)**
- The Iterator interface has three methods for manipulating elements in a Collection (List or LinkedList):
  – hasNext(), next() and remove()
- Found in the java.util package
  import java.util.Iterator;

16 **The Iterator Interface (Page 2)**
- The Iterator interface has three methods for manipulating Collection elements:
  – hasNext(), next() and remove()
- Format to instantiate:
  Iterator<Type> *object* = *collectionObject*.iterator();
  – *Type* is the generic type
- Example:
  Iterator<Double> iterator = list.iterator();

17 **The iterator() Method**
- The iterator() method of the List class returns an iterator over the elements in this collection (points to the elements in sequence)
- Format:
  collectionObject.iterator()
- Example:
  Iterator<Double> iterator = list.iterator();

18 **The hasNext() Method**
- The hasNext() method of the Iterator interface returns a boolean value indicating whether or not the iteration has more elements
- Format:
  iteratorObject.hasNext()
- Example:
  while (iterator.hasNext())

19 **The next() Method**
- The next() method of the Iterator interface returns the next element in the iteration (List)
- Format:
  iteratorObject.next()
- Example:

if (removeList.contains(iterator.next()))

20 **The remove() Method**

- The remove() method of the Iterator interface removes from the underlying List the last element returned by this iterator
- Can be called only once per call to next()
- Format:
  iteratorObject.remove()
- Example:
  iterator.remove();

21 **The contains() Method**

- The contains() method of the Collection interface returns a boolean value that indicates if this collection contains the specified element
- Format:
  collectionObject.contains()
- Example:
  if (removeList.contains(iterator.next()))

23 **The LinkedList Class (Page 1)**

- The LinkedList class implements a *linked list* data structure
- In computer science, a linked list is a linear collection of data elements called nodes pointing to the next node by means of pointer
- Found in the java.util package
  import java.util.LinkedList;

25 **The LinkedList Class (Page 2)**

- Format to instantiate:
  LinkedList<Type> *object* = new LinkedList();
  – *Type* is the generic type
- Examples:
  LinkedList<String> list1 = new LinkedList();
  List<String> list2 = new LinkedList();

26 **The addAll() Method**

- The addAll() method of the LinkedList class appends all of the elements in the specified collection to the end of this list
- Format:
  linkedListObject.addAll(*collection*)
- Example:
  list1.addAll(list2);

27 **The ListIterator Interface (Page 1)**

- An iterator for LinkedLists that allows:
  – Traversing the list in either direction

– Modifying the list during iteration
– Obtaining the iterator's current position in the list

- It is a *subinterface* of Iterator and includes all of that interface's methods:
  – hasNext(), next() and remove()
- Found in the java.util package
  import java.util.ListIterator;
-

28 **The ListIterator Interface (Page 2)**

- The ListIterator interface "inherits" the methods from Iterator and has additional methods for manipulating elements in a LinkedList
- Format to instantiate:
  ListIterator<Type> *object* = *linkedListObject*.listIterator();
  – *Type* is the generic type
- Example:
  ListIterator<String> iterator = list1.listIterator();

30 **The listIterator() Method (Page 2)**

- Format:
  listObject.listIterator([*position*])
  – *position* indicates the starting position in the list
- Examples:
  ListIterator<String> iterator = list.listIterator();
  ListIterator<String> iterator = list.listIterator(list.size());

31 **The subList() Method**

- The subList() method of the List interface returns a view of a portion of the list
- Format:
  list.subList(*fromIndex*, *toIndex*)
  – *fromIndex* is the starting location in the list inclusive
  – *toIndex* is the ending location in the list exclusive
- Example:
  list.subList(4, 7).clear();

32 **The clear() Method**

- The clear() method of the List interface removes all the elements from the list or sublist
- Format:
  list.clear()
- Example:
  list.subList(4, 7).clear();
  – In combination with method sublist() removes only the 4th, 5th and 6th elements from the List object

33 **The hasPrevious() Method**

- The hasPrevious() method of interface ListIterator returns a boolean value indicating whether or not the iteration has more elements
- Format:
  iteratorObject.hasPrevious()
- Example:
  while (iterator.hasPrevious())

34 **The previous() Method**

- The previous() method of the ListIterator interface returns the previous element in the LinkedList (collection)
- Format:
  iteratorObject.next()
- Example:
  if (collection2.contains(iterator.next()))

35 **The set() Method**

- The set() method of the ListIterator interface replaces the last element returned by the most recent next() or previous() method
- Format:
  listIteratorObject.set(*element*)
- Example:
  iterator.set(state.toUpperCase());

37 **The Arrays Class**

- The Arrays class (not Array class) contains a set of static methods for manipulating arrays
- E.g. sorting and searching
- Format:
  Arrays.*arraysMethod*(*array*)
- Example:
  List<Double> links = Arrays.asList(numbers);

38 **The asList() Method (Page 1)**

- The asList() method of class Arrays returns the elements of an array converted to a List type
- Format:
  Arrays.asList(*array*)
- Example:
  List<Double> list = Arrays.asList(numbers);

39 **The asList() Method (Page 2)**

- More examples:
  LinkedList<Double> links = new LinkedList(Arrays.asList(numbers));

– As the argument to the LinkedList constructor
LinkedList<Double> links = (LinkedList) Arrays.asList(numbers);
– Cast to type LinkedList

40 **The add() Method with Index**

- The overloaded add() method of the LinkedList class provides an *index* that inserts the element at the specified position in the list
- Format:
  linkedListObject.add(*index*, *element*)
- Example:
  links.add(2, 4.0);

41 **The addFirst() Method**

- The addFirst() method of the LinkedList class inserts the specified *element* at the beginning the list
- Format:
  linkedListObject.addFirst(*element*)
- Example:
  links.addFirst(1.5);

42 **The addLast() Method**

- The addLast() method of the LinkedList class inserts the specified *element* to the end of the list
- Format:
  linkedListObject.addLast(*element*)
- Example:
  links.addFirst(10.0);

43 **The toArray() Method**

- The toArray() method of the LinkedList class returns an array of the specified size from the linked list
- Format:
  genericCollection.toArray(new GenericType[*size*])
  – *size* if optional
- Example:
  double[] numbers = links.toArray(new Double[links.size()]);

45 **The Collections Class (Page 1)**

- The Collections class (not Collection interface) contains several static methods for manipulating elements in collections
  – E.g. sort(), reverse(), shuffle(), binarySearch(), remove(), etc.
- Found in the java.util package
  import java.util.Collections;

46 **The Collections Class (Page 2)**

- Format:
  Collections.*method*(*collection*);
- Example:
  Collections.sort(list);

47 **The sort() Method (Page 1)**

- The sort() method of class Collections can sort a list in either ascending or descending order
- Format ascending order:
  Collections.sort(*collection*);
- Example:
  Collections.sort(list);

48 **The sort() Method (Page 2)**

- To sort the collection in descending order pass method reverseOrder() as the second argument to the method
- Format ascending order:
  Collections.sort(*collection*, Collections.reverseOrder());
- Example:
  Collections.sort(list, Collections.reverseOrder());

50 **The Comparator Interface (Page 1)**

- The Comparator interface is used to create classes that sort the objects of user-defined class
- Found in the java.util package
  import java.util.Comparator;

51 **The Comparator Interface (Page 2)**

- The interface provides the method compare() used when two or more objects from the same class are compared
- Classes that implement Comparator can impose a "total ordering" on the entire collection of objects instantiated from the same class

52 **The Comparator Interface (Page 3)**

- Format to create a Comparator class:
  public/private *ClassName* implements Comparator<Type> { … }
  – *Type* is the generic type
- Example:
  public AgeComparator implements Comparator<Age> { … }

53 **The compare() Method (Page 1)**

- The compare() method for an object that implements Comparator compares its two arguments for order
  – The method returns a *positive integer*, *zero* or a *negative integer* depending if the first argument is greater than, equal to, or less than the second

54 **The compare() Method (Page 2)**

- It even is possible to *sort on multiple fields* from the objects by separate returns from separate instance variable comparisons
  – Sorting on major sort field, intermediate sort field(s), minor sort field, e.g. *lastName*, *firstName*, *middleInitial*
- Used in combination with method sort() of the Collections class, can impose *total ordering* on the entire collection of objects

55 **The compare() Method (Page 3)**

- Example:

```
public int compare(Age age1, Age age2)
{
   int yearsDifference = age1.getYears() - age2.getYears();

   if (yearsDifference != 0)
   {
      return yearsDifference;
   }

   int monthsDifference = age1.getMonths() - age2.getMonths();
   return monthsDifference;
}
```

56 **Method sort() with a Comparator**

- The sort() method of class Collection may take an optional second argument which is a new Comparator object containing sort instructions
- Format ascending order:
  Collections.sort(*collection*, new *ComparatorConstructor*());
  – *ComparatorConstructor* is from the class that implements Comparator<*Type*>
- Example:
  Collections.sort(list, new AgeComparator());
-

58 **Enum Types (Page 1)**

- An enum type is a special data type that enables for a variable to be a set of predefined constants
- Common examples include compass directions (values of NORTH, SOUTH, EAST, and WEST) and the days of the week
- Because they are constants, the names of an enum type's fields often are in uppercase letters (although this is not absolute)

59 **Enum Types (Page 2)**

- Format to declare:

```
public/private enum EnumName
{
   Constant1, Constant2, …
}
```

- Example:

```
public enum Suit
{
   Clubs, Diamonds, Hearts, Spades
}
```

60 **Enum Types (Page 3)**

- Because an enum type is a collection, it is possible to iterate though the elements:

```
for (Suit suit : Suit.values() )
{
   System.out.println(suit);
}
```

– For the enum on the previous page prints:

```
Clubs
Diamonds
Hearts
Spades
```

61 **The values() Method**

- The values() method of an enum type returns the *entire list* of all values from the enum
- Format:
  enum.values()
- This collection can be accessed one element at a time in a for-each loop:

```
for (Suit suit : Suit.values() )
{
   System.out.println(suit);
}
```

62 **The shuffle() Method**

- The shuffle() method of class Collections *randomly* rearranges ("shuffles") the list
- Format ascending order:
  Collections.shuffle(*collection*);
- Example:

```
Collections.shuffle(list);
```

64 **The reverse() Method**

- The static method reverse() of class Collections reverses the order of the elements in the argument *list*

- Format:
  Collections.reverse(*collection*);
- Example:
  Collections.reverse(list);

65 **The copy() Method**

- The static method copy() of class Collections copies all of the elements from the *destination list* into the *source list*
- The destination list must be at least as long as the source list or remaining elements are unchanged
- Format:
  Collections.copy(*destination*, *source*);
- Example:
  Collections.copy(list, copyList);

66 **The fill() Method**

- The static method fill() of class Collections all elements in the first argument *list* with the value of the second argument *object*
- Format:
  Collections.fill(*collection*, *object*);
- Example:
  Collections.fill(list, 3.3);

67 **The max() Method**

- The static method max() of class Collections returns the *maximum* element of the given collection of the argument *list*
- Format:
  Collections.max(*collection*);
- Example:
  Collections.max(list);

68 **The min() Method**

- The static method min() of class Collections returns the *minimum* element of the given collection of the argument *list*
- Format:
  Collections.min(*collection*);
- Example:
  Collections.min(list);

70 **The binarySearch() Method (Page 1)**

- The binarySearch() method of class Collections searches the list for the specified object (*key*) using the binary search algorithm
  – More efficient than a sequential search
- It returns the index (an int) of the matching element or minus one (-1) or other

negative value if the search fails
- The list must be *sorted* into ascending order according prior to making the call

71 **The binarySearch() Method (Page 2)**
- Format:
  Collections.binarySearch(*collection*, *key*);
- Example:
  String key = textFieldSearch.getText();
  int result = Collections.binarySearch(list, key);

73 **The disjoint() Method**
- The disjoint() method of class Collections returns a boolean value true if the two lists have *no elements* in common
- Format:
  Collections.disjoint(*collection1*, *collection2*);
- Example:
  boolean disjoint = Collections.disjoint(list1, list2);

74 **The addAll() Method**
- The addAll() method of class Collections adds all of the specified elements to the end of list
- Format:
  Collections.addAll(*destinationList*, *elementOrList1*, *elementOrList2*, ...);
  – *elementOrList* may be a single element or another list
- Examples:
  Collections.addAll(list2, numbers);
  Collections.addAll(flavors, "Vanilla", "Chocolate", "Strawberry", flavors2);

75 **The frequency() Method**
- The frequency() method of class Collections returns an int which is the count of a specific element (value) in the list
- Format:
  Collections.frequency(*collection*, *element*);
  – *element* is the value counted in the list
- Examples:
  int frequency = Collections.frequency(list2, 2.0);

77 **The Queue Interface (Page 1)**
- The Queue interface is a subtype of and derives from the Collection interface
- Queues typically, but do not necessarily, order elements in a FIFO (first-in-first-out) manner
  – Among the exceptions are *priority queues*, which order elements according to a supplied comparator, or the elements' natural ordering
- Found in the java.util package

import java.util.Queue;

78 **The Queue Interface (Page 2)**

- Format to instantiate a Queue from a PriorityQueue constructor:
  Queue<*Type*> *object* = new PriorityQueue();
  – *Type* is the generic type
- Example:
  Queue<String> queue = new PriorityQueue();

79 **The PriorityQueue Class (Page 1)**

- A PriorityQueue is a queue in which elements are ordered according to their *natural ordering*
- Found in the java.util package
  import java.util.PriorityQueue;

80 **The PriorityQueue Class (Page 2)**

- Like a normal Queue object, queue elements are removed from the head (front) of the line
- Unlike a normal Queue object, queue elements are inserted into the queue by their sequential value (not at the tail)

81 **The PriorityQueue Class (Page 3)**

- Format to instantiate a Queue from a PriorityQueue constructor :
  Queue<*Type*> *object* = new PriorityQueue();
  – *Type* is the generic type
- Example:
  Queue<String> queue = new PriorityQueue();

82 **The offer() Method**

- The offer() method inserts an element into the priority queue in *sequential order*
- Format:
  priorityQueueObject.offer(*element*);
  – *element* is the element to be inserted
- Example:
  queue.offer("Alex");

83 **The peek() Method**

- The peek() method retrieves, but does not remove, the head (first element) in the priority queue
  – Returns null if the queue is empty
- Format:
  priorityQueueObject.peek()
- Example:
  System.out.println(queue.peek());

84 **The poll() Method**

- The poll() method retrieves and removes the head (first element) in the priority queue
  – Returns null if the queue is empty
- Format:
  priorityQueueObject.poll()
- Example:
  queue.poll();

85 **The size() Method**

- The size() method returns the number of elements in this priority queue as an int
- Format:
  priorityQueueObject.size()
- Example:
  while (queue.size() > 0)

87 **The Set Interface (Page 1)**

- The Set interface is a subtype of and derives from the Collection interface
- It is collection that contains no duplicate elements and as such it models the mathematical *set* abstraction
- Found in the java.util package
  import java.util.Set;

88 **The Set Interface (Page 2)**

- Format to instantiate a Set from a HashSet constructor:
  Set<*Type*> *object* = new HashSet();
  – *Type* is the generic type
- Example:
  Set<Integer> set = new HashSet();

89 **The HashSet Class (Page 1)**

- The class HashSet implements both the Set and Collection interfaces
- It stores its values without regard to a specific order
- Found in the java.util package
  import java.util.HashSet;
-

90 **The HashSet Class (Page 2)**

- Format to instantiate a Set from a HashSet constructor:
  Set<*Type*> *object* = new HashSet();
  – *Type* is the generic type
- Example:
  Set<Integer> set = new HashSet();

-

92 **The SortedSet Interface (Page 1)**

- The SortedSet interface is a subtype of and derives from the Set interface
- It is set that contains no duplicate elements and further provides a *total ordering* on its elements
- Found in the java.util package
  import java.util.SortedSet;

93 **The SortedSet Interface (Page 2)**

- Format to instantiate a SortedSet from a TreeSet constructor:
  SortedSet<*Type*> *object* = new TreeSet();
  – *Type* is the generic type
- Example:
  SortedSet<Integer> tree = new TreeSet();

94 **The TreeSet Class (Page 1)**

- The class TreeSet implements the SortedSet, Set and Collection interfaces
- It stores values in "sorted order" using a *binary search tree* data structure
  – Binary search trees have a *root* (balance point of tree)
  – All nodes lower than the root make up the *head set*
  – The root and all nodes greater than the root make up the *tail set*
- Found in the java.util package
  import java.util.TreeSet;
-

96 **The TreeSet Class (Page 2)**

- Format to instantiate a Set from a HashSet constructor:
  Set<*Type*> *object* = new HashSet();
  – *Type* is the generic type
- Example:
  Set<Integer> set = new HashSet();
-

97 **The headSet() Method**

- For a TreeSet object, the headSet() method returns the *head set* of a binary search tree from the specified root
- Format:
  treeSetObject.headSet(*root*)
- Example:
  System.out.println(tree.headSet(8));

98 **The tailSet() Method**

- For a TreeSet object, the tailSet() method returns the *tail set* of a binary search tree from the specified root

- Format:
  treeSetObject.tailSet(*root*)
- Example:
  System.out.println(tree.tailSet(8));

100 **The Map Interface (Page 1)**

- The Map interface creates objects that map *keys* to *values*.
- A map cannot contain duplicate keys, but each key can map to at most one or more values
- Found in the java.util package
  import java.util.Map;

101 **The Map Interface (Page 2)**

- Format to instantiate a Map from a HashMap constructor:
  Map<*Key, Value*> *object* = new HashMap();
  – The keys are created in hashed (no specific) order
  – *Type* is the generic type
- Example:
  Map<String, Integer> map1 = new HashMap();
-

102 **The Map Interface (Page 3)**

- Format to instantiate a Map from a TreeMap constructor:
  Map<*Key, Value*> *object* = new TreeMap();
  – The keys are created in "sorted order" using a binary search tree
  – *Type* is the generic type
- Example:
  Map<String, Integer> map2 = new TreeMap();
-

103 **The Map Interface (Page 4)**

- Format to instantiate a Map from a LinkedHashMap constructor:
  Map<*Key, Value*> *object* = new LinkedHashMap ();
  – The keys are created in sequence in the order in which they were added
  – *Type* is the generic type
- Example:
  Map<String, Integer> map3 = new TreeMap();
-

104 **The put() Method**

- The put() method for a Map object adds a new element (key and value) to the map
- Format:
  mapObject.put(*key*, *value*)
- Example:

map.put(word, 1);

105 **The get() Method**

- The get() method for a Map object returns the value that matches the key that is given as an argument
- Format:
  mapObject.get(*key*)
  – *key* is a "search key" possibly from an input source
- Example:
  System.out.println(map.get(key));

107 **The containsKey() Method**

- The containsKey() method for a Map object is a boolean method that indicates if the key argument is found in the list of keys of the map
- Format:
  mapObject.containsKey(*key*)
- Example:
  if (map.containsKey(word)) { … }

108 **The size() Method**

- The size() method for a Map object returns an int which is the number of elements in the map
- Format:
  mapObject.size()
- Example:
  System.out.println(map.size());

109 **The isEmpty() Method**

- The isEmpty() method for a Map object is a boolean method that returns a value which indicates whether the map contains any elements
- Format:
  mapObject.ifEmpty()
- Example:
  if (map.isEmpty()) { … }