Family Name: ..... **Other Names:** .....

Student ID: ..... **Signature** .....

COMP 103: Final Test

2021, Oct 28 ** WITH SOLUTIONS **

Instructions

- Time allowed: **TWO HOURS**
- Attempt ALL Questions.
- The examination will be marked out of 120 marks.
- Brief Documentation is at the end of the examination script
- Answer in the appropriate boxes if possible — if you write your answer elsewhere, make it clear where your answer can be found.
- There are spare pages for your working and your answers in this examination, but you may ask for additional paper if you need it.

Questions	**Marks**	
1. Properties of Collections	[14]	
2. Lists, Maps, and Comparable	[30]	
3. Simulation with Collections	[25]	
4. Traversing General Trees	[28]	
5. Complexity: Big-O costs	[13]	
6. Traversing Graphs	[10]	
	TOTAL:	

Date of revision: October 28, 2021

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

Question 1. Properties of Collections **[14 marks]**

For each question (a) to (c), say which of the collection types (List, Stack, Queue, Set, Collection) have the specified properties.

(a) **[2 marks]** Items can be added at the front/top of the collection:

List, Stack

(b) **[2 marks]** The collection guarantees to keep items in the collection in the order determined by how they were added to the collection:

List, Stack, Queue

(c) **[2 marks]** The collection will not allow duplicate items in the collection.

Set

(d) **[4 marks]** For a TreeSet, state whether each property is true or false (circle the answer).

true/false : Items must have a natural ordering to be stored in the collection.

true/false : Either Items have a natural ordering, or the TreeSet must be created with a comparator.

true/false : The cost of adding an item is independent of the size of the set.

true/false : The cost of adding an item doubles if the set doubles in size

true/false : A foreach loop through the set will enumerate the items in the natural order of the items (or the order of the comparator)

(e) **[4 marks]** For a HashMap, state whether each property is true or false (circle the answer).

true/false : Values must be added with a key

true/false : The cost of accessing the value associated with a key is independent of the size of the Map

true/false : Two different keys must have different values associated with them

true/false : Adding a new value associated with a key will delete any previous value associated with the key.

true/false : A foreach loop through the keys of the map will always step through the keys in their natural order.

Question 2. Lists, Maps, and Comparable **[30 marks]**

For this question, you must complete some methods in a program for an adventure-style multi-player video-game. The game involves finding and using Items in the game world.

Each item in the game has a unique identifier (a String), and has a name (eg, `"green potion"`), a status (active or inactive), and a list of Strings giving its characteristics, such as `"poison"` and `"locked"`.

Documentation of the methods of the Item class:

```
Item class:
    public String getID();                          // returns the identifier of the item
    public String getName();                        // returns the name of the item
    public void setActive(boolean status);          // sets the item to be active (if status is true)
                                                    // or inactive (if status is false)
    public List<String> getCharacteristics();       // returns a list of the item's characteristics
                                                    // (guaranteed to be not null)
```

The game program has a field that contains a Map of all the Items, indexed by their identifier:

```
    public Map<String,Item> allItems;
```

There is a diagram of an example allItems Map on the facing page.

(a) **[6 marks]** Complete the following findItems(...) method which will return a set of the identifiers of all the Items in allItems that have the given characteristic.

For example, findItems("gold") would return the set {"i1021", "i1833"} on the Map on the facing page.

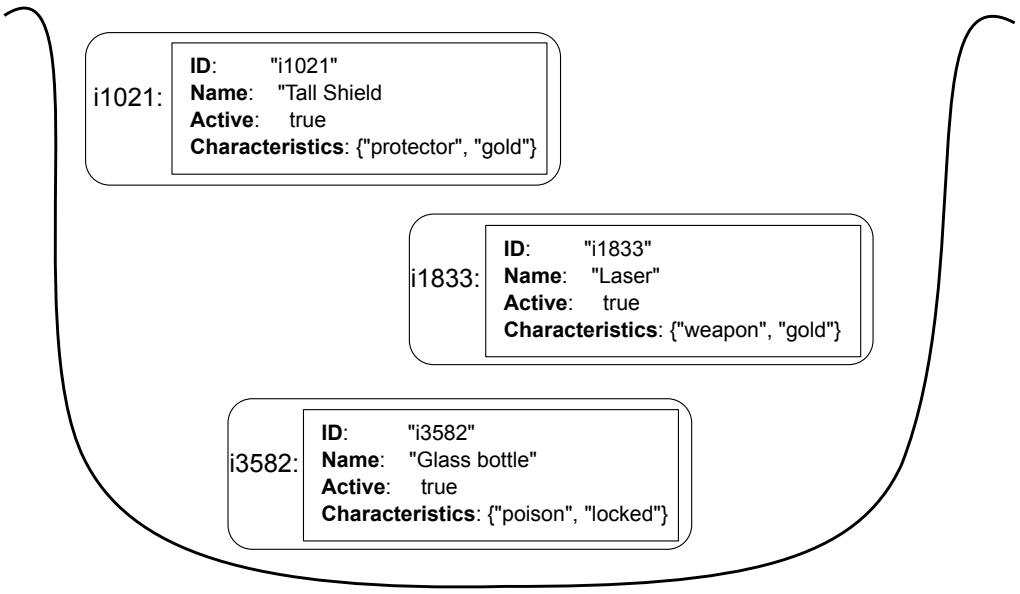
```
    public Set<String> findItems(String characteristic){
        Set<String> answer = new HashSet<String>();
        for (Item item : allItems.values()){
            if (item.getCharacteristics().contains(characteristic)){
                answer.add(item.getID());
            }
        }
OR      for (String ID : allItems.keySet()){
            if (allitems.get(ID).getCharacteristics().contains(characteristic)){
                answer.add(ID);
            }
        }
        return answer;
    }
```

(Question 2 continued on next page)

(b) **[4 marks]** Briefly explain why representing the characteristics in a List is a bad idea and suggest a better option.

A list would allow a characteristic to be duplicated, but it makes no sense to have duplicate properties.
Searching a list for a characteristic may be slow
A better option would be to use a HashSet of Strings

Example of allItems Map containing three Items, indexed by their ID's:



(Question 2 continued on next page)

(Question 2 continued)

The game program also contains a list of the current players, represented by Player objects:

```
public List<Player> currentPlayers;
```

A Player object records lots of information about a player, including the collection of ids of the items that the player is currently carrying around.

Documentation of some methods of the Player class:

```
Player class:
    public String getName();              // returns the name of the player
    public Set<String> getItems();        // returns the Set of identifiers of the items
                                          //  that the player is carrying.
    public void removeItem(String id);    // remove the specified item from the list
                                          //  of items that the player is carrying.
    public double getHealth();            // returns the current health status of the player
    public void setSleep();               // sets the player to be asleep
```

An example of a Player object might be

Name: "Jamie"
Health: 58
Sleep: false
ItemIDs: {"i1021", "i3582", "i9884"}

(c) **[6 marks]** Complete the following putToSleep(...) method which should set the given player to be asleep and also set all the items they are carrying to be inactive.

Note: it is possible for a Player to still have the identifier of an item that has been destroyed and is no longer in the allItems Map. For example, the ID "i9884" in the Player above is not in the map on the previous page.

```
public void putToSleep(Player player){
    player.setSleep();
    for (String itemId : player.getItems()){
        Item item = allItems.get(itemId);
        if (item != null){
            item.setActive(false);
        }
    }




}
```

(Question 2 continued on next page)

(Question 2 continued)

(d) **[6 marks]** [Harder!] Complete the following cleanItemLists() method which should check the list of items carried by each player and remove any identifiers for items that have been destroyed and are no longer in the allItems Map.

```
public void cleanItemLists (){
    for (Player player : currentPlayers ){
        List<String> destroyedIDs = new ArrayList<String>();
        for (String itemId : player.getItems()){
            if (! allItems .containsKey(itemId)){ // or (allItems .get(itemId)==null)
                destroyedIDs.add(itemId );
            }
        }
        for (String id : destroyedIDs){
            player .removeItem(id);
        }
    }

}
```

(Question 2 continued on next page)

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

(Question 2 continued)

(e) **[8 marks]** It has been decided that the Item class needs to implement the Comparable interface (see documentation sheet). The natural order of the class should be based on the id field (since that is unique to each item) using normal alphabetical order.

The revised Item class should also work correctly with both TreeSets and HashSets: *i.e.,* hash code and comparisons should be consistent.

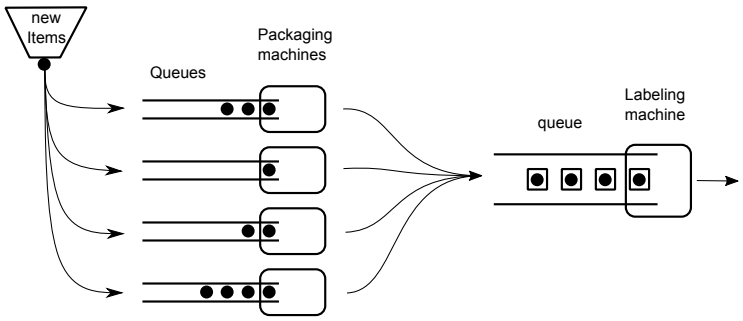
Complete the equals(...), compareTo(...) and hashCode() methods for the Item class below.

```
public class Item implements Comparable <Item> {

    private String id;      // a unique identifier for the item.
    ... // other fields

    public String getID(){return id;}
    ... // other methods

    public boolean equals(Object other){
        if(this == other) return true;
        if(other == null) return false;
        if(!(other instanceof Item)) return false;

        return this.id.equals(((Item)other).id);









    }

    public int compareTo(Item other) {
        return id.compareTo(other.id);


    }
    public int hashCode() {
        return id.hashCode();


    }


}
```

Question 3. Simulation with Collections **[25 marks]**

Suppose you are writing a program to simulate the operation of an automated warehouse with a collection of packaging machines that can put items in boxes. Each machine has a queue of Item to process, and each item may take several time ticks to complete.

The warehouse also has a labeling machine that puts labels on boxes. It can label a box in just one time tick.



At each timestep, the program

- Removes the Item at the head of the labeling queue, if there is one, (removeLabeledItem)
- Checks whether there is a new item to package. If so, it adds the item to the shortest packaging machine queue. (enqueueItem)
- Advances the packaging of the Item at the front of each queue by one time "tick" and moves any completed Items from their packaging machine queue to the labeling machine queue. (advanceAllItems)
- draws the state of the queues.

You are to write the removeLabeledItem(), enqueueItem(...), and advanceAllItems() methods.

The WarehouseSimulation class has the following fields, constructor and run() method.

```
public class WarehouseSimulation{
    public static final int N = 4;  // number of packaging machines
    private List<Queue<Item>> packingQueues = new ArrayList<Queue<Item>>();
    private Queue<Item> labelingQueue = new ArrayDeque<Item>();

    public WarehouseSimulation(){
      for (int i=0; i<N; i++){
         packingQueues.add(new ArrayDeque<Item>());
      }
    }
    public void run (){
        int time = 0;
        while (true){
            time++;
            removeLabeledItem();                          // subquestion (a)
            Item item = getNewItem();
            if (item != null){ enqueueItem(item); }       // subquestion (b)
            advanceAllItems();                            // subquestion (c)
            drawQueues();
        }
    }
}
```

(Question 3 continued on next page)

(Question 3 continued)

The Item class has the following methods:

```
Item class:
    public void advancePackagingByTick();
    public boolean completed();
```

(a) **[3 marks]** Complete the following removeLabeledItem() method which should remove the Item at the head of the labelingQueue, if there is one.

```
public void removeLabeledItem(){
    labelinqQueue.poll();



}
```

(b) **[10 marks]** Complete the following enqueueItem(...) method which should add the given item to the shortest packing queue.

```
public void enqueueItem(Item item){
    Queue<Item> shortestQueue = packingQueues.get(0);
    for (Queue<Item> queue : packingQueues){
        if (queue.size()<shortestQueue.size()){
            shortestQueue = queue;
        }
    }
    shortestQueue.offer(item);









}
```

(Question 3 continued on next page)

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

(Question 3 continued)

(c) **[12 marks]** Complete the following advanceAllItems() method which should advance the packaging of the Item at the front of each packing queue by one time "tick", and move any completed Items to the labeling queue.

```
public void advanceAllItems(){
    for(Queue<Item> queue : packingQueues){
        if (!queue.isEmpty()){
            Item item = queue.peek();
            item.advancePackagingByTick();
            if (item.completed()) {
                queue.poll();
                labelingQueue.offer(item);
            }
        }
    }









}
```

Question 4. Traversing General Trees **[28 marks]**

This question concerns a PC configurator that manages the components in a PC, represented by Component objects.

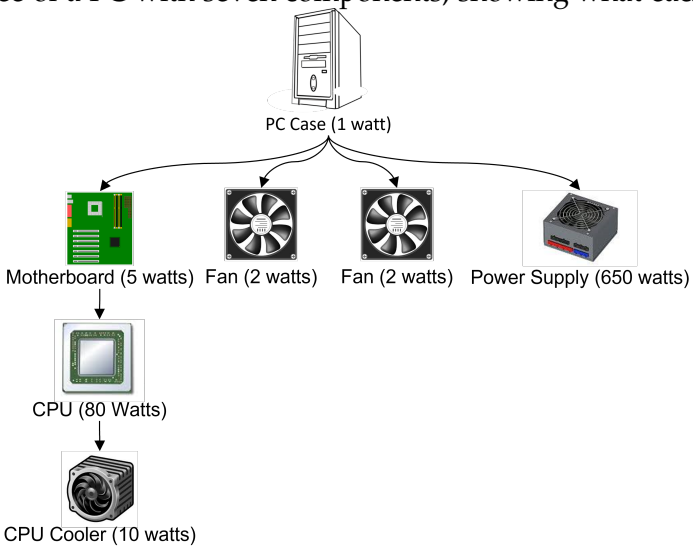
Each Component has a name, wattage and a level. The methods in the Component class include the following:

```
Component class:
    public String getName();     // returns the name of the component (e.g., "Fan")
    public int getWatts();       // returns the wattage associated with the component
    public int getLevel();       // returns the mounting level of the component
    public String toString();    // returns a String describing the component,
                                 // e.g. "Fan (2 watts)" for a fan requiring 2 watts
```

The wattage of a component is its power consumption, except for components named "Power Supply" in which case the wattage is the maximum power it can provide.

In a PC, some of the components are mounted on other components. For example, the CPU is usually mounted on the Motherboard, which is mounted on the PC Case.

Here is an example tree of a PC with seven components, showing what each component is mounted on:



The configurator program represents the collection of Components in a PC as a general tree of Component objects using GTNode objects to represent the tree nodes. The methods in the GTNode class include the following:

```
GTNode<E> class:
    public E getItem();                    // return item in the node
    public int numChildren();              // return number of children of the node
    public GTNode<E> getChild(int i);      // return i'th child
```

Note: This version of GTNode is **not** Iterable; to iterate through the children of a node, use:

```
for ( int i=0; i<node.numChildren(); i++){... node.getChild(i) ...}
```

(Question 4 continued on next page)

(Question 4 continued)

(a) **[10 marks]** Complete the following printComponents(...) method which is given the root node of a component tree, and prints out all the components in the tree, using indentation to show the structure.

For example, the tree on the previous page should be printed as:

```
Case (1 watts)
    Motherboard (5 watts)
        CPU (80 watts)
            CPU Cooler (10 watts)
    Fan (2 watts)
    Fan (2 watts)
    Power Supply (650 watts)
```

Hint: You may use a "helper function" that takes more arguments.
Note: You can still get 7 of the marks for printing all the components in the tree without indentation

```
public void printComponents(GTNode<Component> root) {
    printComponentsWithIndent(root, "");
}
public void  printComponentsWithIndent(GTNode<Component> node, String indent) {
    System.out. println (indent + node.getItem());

    for (int i=0; i<node.numChildren(); i++)
        printComponentsWithIndent(node.getChild(i), indent + "        ");
}
}
```

(Question 4 continued on next page)

(Question 4 continued)

(b) **[9 marks]** Complete the following powerUse(...) method which should return the total wattage of a tree of Components, **except the component named "power supply"**. For the example tree above, the method should return 100 ($1 + 5 + 80 + 10 + 2 + 2 = 100$).

```
public int powerUse(GTNode<Component> node) {
    int power = 0;
    Component comp = node.getItem();
    if (!comp.getName().equals("Power Supply")){
        power = comp.getWatts();
    }
    for ( int i=0; i<node.numChildren(); i++){
        power += powerUse(node.getChild(i));
    }
    return power;
}
```

(Question 4 continued on next page)

(Question 4 continued)

(c) **[9 marks]** [Harder!] Each component has a level that constrains where a component can be mounted. Every component must be mounted on a component with a lower level.

For example, if the CPU cooler is level 4, then it must be mounted on a component that is level 3 or less.

Complete the following checkLevels(...) method which returns true only if every component in a tree has a lower level than all its child components.

Note: The component level values may not be the same as the depth in the tree, since a level 4 component can be mounted on a level 1 component.

```
public boolean checkLevels(GTNode<Component> node) {
    int level = node.getItem().getLevel();

    for (int i=0; i<node.numChildren(); i++) {
        if (node.getChild(i).getItem().getLevel() <= level)
            return false;

        if (!checkLevels(node.getChild(i)))
            return false;
    }

    return true;









}
```

Question 5. Complexity: Big-O costs **[13 marks]**

For each question below, work out the cost (in Big-O notation) by

- working out the cost of performing each line once.
- working out the number of times each line will be performed.
- computing the total cost.

(a) **[4 marks]** What are the Big-O costs of the fragments of code below. Assume the size of list is n.

```
for (int i = 0; i <= list.size(); i++) {
    Item item = list.remove(i);          // cost = O( n ) times= n
    list.add(item);                      // cost = O( 1 ) times= n
}

// Total Cost = O( n^2 )
```

(b) **[4 marks]** What are the Big-O costs of the fragments of code below (in the worst case)? Assume the size of the list is n.

```
int matches = 0;
for (Item a : list){
   for (Item b : list ) {
       if (a.equals(b)){                 // cost = O( 1 ) times= n^2
          matches++;                     // cost = O( 1 ) times= n^2
       }
   }
}
matches = matches - list.size();         // cost = O( 1 ) times= 1

// Total Cost = O( n^2 )
```

(Question 5 continued on next page)

(c) **[5 marks]** A program has a TreeMap of Persons indexed by names.

When the TreeMap contains 1,000,000 Persons (approximately 2^{20}), the program takes 60 microseconds to look up a name in the TreeMap document.

If the TreeMap had 8,000,000 Persons, how long would you expect the program to take to look up a name in the TreeMap? Explain why.

About 69 microseconds
Finding a value by key in a TreeMap is O(lg(n))
Find a Person in this map typically takes 20 steps,
which is 3 microseconds per step
With 8,000,000 Persons, the program will have to search 3 more steps
which is 9 more microseconds

Question 6. Traversing Graphs **[10 marks]**

"Island Hoppers Inc." is an airline that services islands within a region of the Pacific. Each island that can be reached from another island with a direct flight is considered a "neighbour" of that island. There are no one-way connections, so if there is a flight from X to Y then there is also a flight from Y to X.

"Island Hoppers Inc." use software which features an Island class with the following methods:

```
Island class:
    public String getName();                  // returns the name of the island
    public void addNeighbour(Island n);       // add n as a possible destination
}
```

Note: you can use a foreach loop to iterate through the neighbours of an island:

```
for ( Island neighbour : island ){...}
```

(a) **[2 marks]** Complete the following addConnection(...) method in the Island class that establishes a flight connection between two islands.

```
public void addConnection(Island n) {


  this.addNeighbour(n);

  n.addNeighbour(this);




}
```

(Question 6 continued on next page)

(Question 6 continued)

(b) **[4 marks]** Complete the following accessibleFrom(...) method which returns the number of islands in the region that that can be accessed from the given island via any number of flights (1, if there are no flights from the island at all).

```
public int accessibleFrom( Island startIsland ) {
    return accessibleFrom( startIsland , new HashSet<Island>());
}

public int accessibleFrom( Island island , Set<Island> visited ) {
    int sum = 1;

    visited .add( island );

    for ( Island neighbour : island )
        if ( ! visited .contains(neighbour))
            sum += accessibleFrom(neighbour, visited );

    return sum;









}
```

(Question 6 continued on next page)

(Question 6 continued)

The company limits the number of flights a pilot can make in one trip.

(c) **[4 marks]** Complete the following islandsWithinHops(...) method which returns a Set of Islands that can be reached from the starting island using no more than the given number of flights.

```
public Set<Island> islandsWithinHops(Island startIsland, int maxhops) {
    return islandsWithinHops(startIsland, maxhops, new HashSet<Island>());
}

public Set<Island> islandsWithinHops(Island island, int hops, Set<Island> visited) {
    visited.add(island);

    if (hops < 1) {
        return visited;
    }
    for (Island neighbour : island) {
        if (!visited.contains(neighbour)) {
            islandsFromWithHops(neighbour, hops-1, visited);
        }
    }
    return visited;









}
```

* * * * * * * * * * * * * * *

Documentation for COMP 103 Exam

Brief, simplified specifications of some relevant Java collection types and classes.

Note: *E* stands for the type of the item in the collection.

```
interface Collection<E>
    public boolean isEmpty()                    // cost: O(1) for standard collection classes
    public int size()                           // cost: O(1) for standard collection classes
    public void clear()
    public boolean add(E item)
    public boolean contains(Object item)
    public boolean remove(Object element)
```

```
interface List<E> extends Collection<E>
    // Implementations: ArrayList
    public boolean isEmpty()
    public int size()
    public void clear()
    public E get(int index)                     // cost: O(1)
    public E set(int index, E element)          // cost: O(1)
    public boolean contains(Object item)        // cost: O(n)
    public void add(int index, E element)       // cost: O(n) (unless index close to end.)
    public E remove(int index)                  // cost: O(n) (unless index close to end.)
    public boolean remove(Object element)       // cost: O(n)
```

```
interface Set extends Collection<E>
    // Implementations: HashSet, TreeSet
    public boolean isEmpty()
    public int size()
    public void clear()
    public boolean add(E item)                  // cost: O(1)       for HashSet
                                                //       O(log(n))  for TreeSet
    public boolean contains(Object item)        // cost: O(1)       for HashSet
                                                //       O(log(n))  for TreeSet
    public boolean remove(Object element)       // cost: O(1)       for HashSet
                                                //       O(log(n))  for TreeSet
```

```
class Stack<E> implements Collection<E>
    public boolean isEmpty()
    public int size()
    pubic void clear()
    public E peek()                 // cost: O(1)
    public E pop()                  // cost: O(1)
    public E push(E element)        // cost: O(1)
    // (peek and pop return null if the queue is empty)
```

```
interface Queue<E> extends Collection<E>
   // Implementations: ArrayDeque, LinkedList, PriorityQueue
   public boolean isEmpty()
   public int size()
   public void clear()
   public E peek()                    // cost: O(1)      for ArrayDeque, LinkedList
                                      //       O(1)      for PriorityQueue
   public E poll()                    // cost: O(1)      for ArrayDeque, LinkedList
                                      //       O(log(n)) for PriorityQueue
   public boolean offer(E element)    // cost: O(1)      for ArrayDeque, LinkedList
                                      //       O(log(n)) for PriorityQueue
   // (peek and poll return null if the queue is empty)
```

```
interface Map<K, V>
   // Implementations: HashMap, TreeMap
   public V get(K key)                // cost: O(1)      for HashMap
                                      //       O(log(n)) for TreeMap
   public V put(K key, V value)       // cost: O(1)      for HashMap
                                      //       O(log(n)) for TreeMap
   public V remove(K key)             // cost: O(1)      for HashMap
                                      //       O(log(n)) for TreeMap
   public boolean containsKey(K key)  // cost: O(1)      for HashMap
                                      //       O(log(n)) for TreeMap
   public Set<K> keySet()             // cost: O(1)
   public Collection<V> values()      // cost: O(1)
   // get returns null if key not present; put & remove return the old value, (if any)
```

```
class Collections
   public void sort(List<E> list);                          // cost = O(n log(n)) in general
                                                            //        O(n)        almost sorted
   public void sort(List<E> list, (E e1, E e2)->{..});      // cost = O(n log(n)) in general
                                                            //        O(n)        almost sorted
   public void swap(List<E> list, int i, int j);            // cost = O(1)
   public void reverse(List<E> list);                       // cost = O(n)
   public void shuffle(List<E> list);                       // cost = O(n)
```

```
interface Comparable<E>          // Items can be compared for sorting or a priority queue.
   public int compareTo(E other); // Comparable objects must have a compareTo method:
    // returns     -ve if this comes before other;
    //             +ve if this comes after other,
    //              0  if this and other are the same
    // Note: The String class is Comparable, and has this method
```

```
interface Iterable<E>            // Can use a foreach loop on these items
   public Iterator<E> iterator(); // Iterable objects must have an iterator method:
```

```
Integer and Double constants:
   Integer.MAX_VALUE; Integer.MIN_VALUE;
   Double.MAX_VALUE; Double.NaN; Double.POSITIVE_INFINITY; Double.NEGATIVE_INFINITY;
```