CSE 12 – Basic Data Structures

Prof. Christine Alvarado

[Slides borrowed/adapted from slides by Cynthia Lee, Rakesh Varna, & Roshni Chandrashekhar]
Announcements

1. GREAT JOB on the midterm. Mean 82, median 86! WOW!
   1. If you didn’t do so well, please come talk to me. There is still time to improve.

2. HW3 is posted. You can start now, but be sure to also do the reading for Friday.
   1. 1-day extension: Due next WEDNESDAY
   2. We will talk about deep and shallow copy on Monday, but understanding the copy constructor is not hard
Today’s topics

- More with Java Generics
- Priority Queues, Trees and Heaps
Java Generics
Life before Generics

List myIntList = new LinkedList();
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.get(0);

Element access needed casting.
Using Generics

List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));
Integer x = myIntList.get(0);

No casting required. Cleaner code.
Life before Generics

List myIntList = new LinkedList();
myIntList.add(new Integer(0));

Does this compile?

A. YES
B. NO
Life before Generics

List myIntList = new LinkedList();
myIntList.add(new Integer(0));
......
......
String s = (String) myIntList.get(0);

Does this compile?
YES! (may be with warnings)

What happens when you run it?
Life before Generics

List myIntList = new LinkedList();
myIntList.add(new Integer(0));
......
......
String s = (String) myIntList.get(0);

Does this compile?
YES! (may be with warnings)

What happens when you run it?
ClassCastException!
Using Generics

List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(new Integer(0));

......
......
String s = (String) myIntList.get(0);

Does this compile?
Using Generics

List<Integer> myIntList = new LinkedList<>();
myIntList.add(new Integer(0));

……
……
String s = (String) myIntList.get(0);

Does this compile?

NO!

Generics => Stronger type checking at compile time =>
No Runtime exceptions
Java Generics

List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());

What does this snippet print?

A. True
B. False

This returns type ‘Class’ not a string.
Java Generics

List <String> l1 = new ArrayList<String> ();
List<Integer> l2 = new ArrayList<Integer> ();
System.out.println(l1.getClass() == l2.getClass());

What does this snippet print?

A. True
B. False

Behavior is same for all classes. Hence the name ‘generic’.

After compilation checks, the generic types are ‘erased’ and replaced by the first "bounding" superclass.

This returns type ‘Class’ not a string. In this case, returns class ArrayList.
public class Node<T> {
    private T data;
    private Node<T> next;

    public Node(T data, Node<T> next) {
        this.data = data;
        this.next = next;
    }
}

public class Node {
    private Object data;
    private Node next;

    public Node(Object data, Node next) {
        this.data = data;
        this.next = next;
    }
}
Non-covariance of Generics

Collection<Animal> animalList;
Collection<Dog> dogList = new LinkedList<Dog>();
animalList = dogList;  // x
...
...
animalList.add(0, new Cat());
Dog d = dogList.get(0);

- Java parameterized types are not covariant! A collection of Dogs is NOT a collection of Animals!

- This means that a Collection (or any class) parameterized by a subclass cannot be assigned to a Collection parameterized by the superclass.
Another example

```java
// Display all Shape objects in the given Collection.
// Call their display() instance method to do that.
static void displayShapes(Collection<Shape> collection){
    for (Shape shape : collection)
        shape.display();
}
...
Collection<Shape> shapes = new LinkedList<Shape>();
shapes.add( new Circle( 5.0 ) );
shapes.add( new Rectangle( 4.5, 21.2 ) );
displayShapes( shapes );

Collection<Circle> circles = new LinkedList<Circle>();
circles.add( new Circle( 5.0 ) );
circles.add( new Circle( 15.0 ) );
circles.add( new Circle( 25.0 ) );
displayShapes( circles );  // ERROR!
```
Yet another example

```java
LinkedList<Integer> intList = new LinkedList<Integer>();
...
... 
Collection<Integer> c = intList;
```

Will this compile?

A. Yes

B. No
Yet another example

```java
LinkedList<Integer> intList = new LinkedList<Integer>();
...
...
Collection<Integer> c = intList;
```

Will this compile?

A. Yes
B. No
Wildcards
Wildcards

Problem: The method should accept a Collection of any subclass of Shape:

```java
static void displayShapes(_______ listOfShapes) {
    for (Shape s : listOfShapes) {
        s.display();
    }
}
```

Java provides a flexible type – the wildcard – `?`
Unbounded wildcard – ‘?’

- `<?>` means *any* type.
- `Collection<?>` is a collection of *any* type.

```java
static void displayShapes(Collection<Shape> listOfShapes) {
    for (Shape s : listOfShapes) {
        s.display();
    }
}
```

Does this solve our problem?
Bounded wildcards

Problem: The method should accept a Collection of any subclass of Shape:

- Unbounded wildcards do not help as they accept a Collection of \textit{any} type.
- What we want is “\textit{any} type that \textit{extends} Shape”
Problem: The method should accept a Collection of any subclass of Shape:

- Unbounded wildcards do not help as they accept a Collection of \textit{any} type.
- What we want is "\textit{any} type that \textit{extends} Shape"

\texttt{< ? extends Shape>}

Accept any type that is 'upper bounded' by Shape
Bounded wildcards

```java
static void displayShapes(Collection<? extends Shape> listOfShapes) {
    for (Shape s : listOfShapes) {
        s.display();
    }
}
```
Problem: `addAll` should accept collections that contain any type that ‘is-a’ `E`.

```
public abstract class AbstractCollection<E> implements Collection<E> {
    // Add all the elements of the argument Collection to this Collection
    public boolean addAll(_______ c) {
        // Add all the elements of the argument Collection to this Collection
    }
}
```

A. `Collections<E>`
B. `Collections<?>`
C. `Collection<?> extends E>`
D. `Collection<?> super E>`
E. More than one / None
public <T extends Comparable<? super T>> void sort(List<T> list);

T must be a type that implements the Comparable interface

What does it have to be able to compare itself to?
Anything that is a T, and all SUPER classes of T

Why superclasses and not subclasses?
Generics... can you do:

Which of the following compile?

- Collection<List> c = new LinkedList<List>();
- LinkedList<List> myL = new LinkedList<List>();
- LinkedList<List> myL = new LinkedList<ArrayList>();
- LinkedList<? extends List> myL = new LinkedList<ArrayList>();
- LinkedList<? super List> myL = new LinkedList<List>();
- LinkedList<? super List> myL = new LinkedList<Collection>();
- LinkedList<Collection> myL = new LinkedList<Collection>();
  myL.add( new ArrayList() );
Generics... can you do:

- Does this compile? If not, why not? And how do you fix it?

```java
public static void myMethod( Collection<List> arg ) { ... }
```

```java
// in main
Collection<LinkedList> myL = new Collection<LinkedList>();
myMethod( myL );
```
Priority Queues
Priority Queue

- Emergency Department waiting room operates as a priority queue
- Patients sorted according to seriousness, NOT how long they have waited
Priority queue implementation options

- **Unsorted linked list**
  - Insert new element in front
  - Remove by searching list for highest-priority item

- **Sorted linked list**
  - Always insert new elements where they go in priority-sorted order
  - Remove from front
Unsorted linked list

- **Add** is **FAST**
  - Just throw it in the list at the front
  - $O(1)$

- **Remove/peek** is **SLOW**
  - Hard to find item the highest priority item—could be anywhere
  - $O(N)$
Priority queue implementations

Sorted linked list

- **Add** is SLOW
  - Need to step through the list to find where item goes in priority-sorted order
  - **O(N)**

- **Remove/peek** is FAST
  - Easy to find item you are looking for (first in list)
  - **O(1)**
Priority queue implementations

We want the best of both

- Fast add AND fast remove/peek
- We will investigate trees as a way to do this
Binary trees
How many of these are valid binary trees?

A. 0-3
B. 4
C. 5
D. 6
E. 7-8
A node object for binary trees

- Similar to a linked list node, it contains a pointer to **data**, and a pointer to the next elements.
- Whereas a linked list node has just one next pointer, a binary node tree has two child pointers, **left** and **right**.
A node object for binary trees

- Similar to a linked list node, it contains a pointer to data, and a pointer to the next elements.
- Whereas a linked list node has just one next pointer, a binary node tree has two child pointers, left and right.
Heaps
Heaps

- Heaps are **one kind** of binary tree
- They have a few special restrictions, in addition to the usual binary tree ADT:
  - Must be **complete**
  - Ordering of data must obey **heap property**
    - Min-heap version: a parent’s data is always \( \leq \) its children’s data
    - Max-heap version: a parent’s data is always \( \geq \) its children’s data
How many of these could be valid heaps?

A. 0-1  
B. 2  
C. 3  
D. 4  
E. 5-8
How many of these are valid min-heaps?

A. 0
B. 1
C. 2
D. 3
In how many places could the largest number in this max-heap be located?

A. 0-2
B. 3-4
C. 5-6
D. 7-8
In how many places could the largest number in this max-heap be located?

A. 0-2  
B. 3-4  
C. 5-6  
D. 7-8

Max heaps are perfect for priority queues, because we always know where the highest priority item is.
In how many places could the number 35 be located in this min-heap?

A. 0-2  
B. 3-4  
C. 5-6  
D. 7-8
In how many places could the number 35 be located in this min-heap?

A. 0-2
B. 3-4
C. 5-6
D. 7-8

Min heaps also good for priority queues, if “high priority” in your system actually means low value (i.e. 1 means most important)
Heap in an array
Heap in an array

- We actually do NOT typically use a node object to implement heaps
- Because they must be **complete**, they fit nicely into an array, so we usually do that
Heap in an array

- For tree of height $h$, array length is $2^h - 1$
- For a node in array index $i$:
  - Parent is at array index:
    - A. $i - 2$
    - B. $i / 2$
    - C. $(i - 1) / 2$
    - D. $2i$
Heap in an array

- For tree of height $h$, array length is $2^h - 1$
- For a node in array index $i$:
  - Left child is at array index:
    - A. $i + 1$
    - B. $i + 2$
    - C. $2i$
    - D. $2i + 1$
Heap in an array

- For tree of height $h$, array length is $2^{h-1}$
- For a node in array index $i$:
  - Parent is at array index: $(i - 1)/2$
  - Left child is at array index: $2i + 1$
  - Right child is at array index: $2i + 2$