JAVA GENERICS

You may have noticed syntax like ‘Position<E>’ and ‘MyHeapEntry<K, V>’ in the stencil and wondered what the E, K and V are ...

These are called **generic** types, and are essentially placeholders for some actual Java object (like a String, or an Integer) that a Position or heap Entry would hold.
WHY GENERICS?

They make your classes more flexible because you can re-use the class to store different type objects.

Imagine if the Java makers implemented an `ArrayList` that can only hold `String`, ex:

```java
public class ArrayList {
    public ArrayList {
        // constructor
    }
    public String add(String s) {
        // some code to add string to list
    }
}
```

Now we have an `ArrayList` who’s add() method can only take in a `String` object.
WHAT ABOUT THINGS BESIDES STRINGS?

But what if I wanted a list of Integers? Booleans? Boxes? Pacman dots?

We’d have to write a new class `ArrayList` whose add() method would take in `that` type of object.

So one solution is defining the add() method as:

```java
public Object add(Object obj) {
    // code to add object to the list
}
```

Since we know that ‘Object’ is a superclass of every Java class, we can now add any type to our list.

Do you foresee any problems with this?
WHAT ABOUT THINGS BESIDES STRINGS?

Here’s an example of our ‘extensible’ `ArrayList`:

```java
ArrayList a = new ArrayList();
a.add(“hello”);
a.add(9578);
a.add(new ArrayList());
a.add(new Comparator());
a.add(3.1459);
```

Now do you see a problem?

If you wanted a list of just one specific type, this polymorphic implementation has no way of enforcing that type-checking.

Which is why … generics!
public class ArrayList<E> {
    public ArrayList() {
        // constructor code
    }
    public E add(E element) {
        // code to add element to list
    }
}

Now how can we use this list?
Say I wanted a list of Integers, I can use this list:

ArrayList<Integer> a = new ArrayList<Integer>();
a.add(1);
a.add(2);
a.add(99999);
SOLUTION WITH GENERICS!

But what if I now did `a.add("hello")`?

The compiler would complain because my list was instantiated to only hold `Integers`.

Now we have an extensible list implementation that can hold any object type, but we limited that extensibility to creation time, so we can enforce type checking on one instance of the list.
SUMMARY OF GENERICS

Remember to implement classes and methods with generic types.

```java
public class MyClass<E> {
    public MyClass() {
        // constructor
    }
}
```

But instantiate them and call methods on specific types.

```java
MyClass<String> theBestObject = new MyClass<String>();
```

Hooray for generics!
COMPARATORS

In the handout, you are told to use a Comparator to compare the values in the heap.

...so what is a Comparator?

It enforces an ordering of things that don’t have an intuitive ordering (like Integers would).
So you could order Strings using a Comparator.

A Comparator is passed into an instance of MyHeap through the constructor.
HOW DO I USE A COMPARATOR?

Comparator is an interface

Aside: What is an interface?

That means every Comparator has the following method:

```java
// Input: Two objects to compare to each other
// Output: A negative int if o1 < o2
//         Zero if o1 = o2
//         A positive int if o1 > o2
public int compare(Object o1, Object o2) {}
```

The Comparator defines what it means to be <, =, and > something
LET’S GET TO HEAP

What is it?
- Minimum key on top
- Binary structure
- Left complete

Diagram:
- 2, C
- 7, T
- 3, H
- 9, P
- 8, M
- 4, L
HOW DO WE MAKE IT?

A binary tree!
LinkedBinaryTree<E>
-Made up of Position<E>

These are Positions
- Make up the structure of the tree
- But Position is an interface...
- If the number of nodes in my tree changes, I need to add/remove these
A heap has key/value pairs -

Entry<K, V>

These are Entries
- Maintain the data of the tree
- If I want to organize the tree, or add/remove data from the tree, I have to deal with these
- But Entry is an interface...
WHAT DO I HAVE TO WRITE?

MyHeap<K, V>
   (next page)

MyLinkedHeapTree<E>
   - Extends NDS4 LinkedBinaryTree<E>
   - The skeleton of your heap
   - Where you want to deal with constant time adding/removing nodes from the structure of the tree
     - Hint hint: your Deque?

MyHeapEntry<K, V>
   - Implements NDS4 Entry<K, V>
   - Stores its key and value... anything else?
WHAT DO I HAVE TO WRITE?

MyHeap<K, V>
- Implements NDS4 AdaptablePriorityQueue<E>
- Where you do your Heap-y things
  - Hint: Does upheaping change the skeleton of the tree?
- It’s adaptable!
  - Can replace the key of entries in the middle of the tree
  - What might I have to do to maintain the organization of the heap?

MyLinkedHeapTree<E>
(previous page)

MyHeapEntry<K, V>
(previous page)
ADD 1

Root: 1

If the deque is empty, set the root to be the new node.

Add the node to the back of the deque.
ADD 2

Root: 1

1

If the parent has no left child, add left child

Add the node to the back of the deque, and check the front of the deque to get the parent node
ADD 3

Root: 1

If the parent has a left child, add right child

Add the node to the back of the deque, and check the front of the deque to get the parent node, and then pop the parent node off front of the deque because it has no space for new child nodes
DELETE 1

Root: 1

1

2

If the parent has a right child, remove right child

Deque

Front

1

2

Back

Pop the node off the back of the deque, and check the parent of the removed node
DELETE 2

Root: 1

1

If the parent has no right child, remove left child

Pop the node off the back of the deque, and check the parent of the removed node
DELETE 3

Root: Null

If the deque is empty, set the root to null.

Pop the node off the back of the deque, the deque will now be empty.
BE CAREFUL WITH `remove()`

- Remember, lecture only covers `removeMin()`, but we are also asking you to implement `remove()`.
- When you insert, you must always upheap. When you `removeMin()`, you must always downheap.
- But what about when you `remove()`? Consider this heap

What if we want to remove this entry?
NDS4!

What is it?

-Your "net.datastructures" for CS16 Java projects!

-A bunch of helpful classes, interfaces, and libraries to help you succeed in all of your endeavors
NDS4 FOR HEAP!

Let’s say I want to use a Deque

Wait! It’s an interface? I can’t make one of those...

Never Fear! “All Known Implementing Classes” is here!
NDS4 FOR HEAP!

LinkedBinaryTree<E> - class

Position<E> - interface
  - Hint: What kind of Positions does LinkedBinaryTree use?

Entry<K, V> - interface

Deque<E> - interface
EXCEPTIONS!

Throughout the project, it is your job to handle what can go wrong and handle the cases appropriately.

This can take a few forms, but the main cases are raising and catching exceptions.

*Note that we mention in the handout and stencil when we require you to handle exceptions*
EXCEPTIONS - RAISING VS. CATCHING

Raising (aka throwing)
You want to raise an exception when you are within a method and know something can go wrong. The exception notifies the calling method that an error occurred.

Example: Getting the min of an empty priority queue

Catching
You want to catch an exception when you are calling a method that raises an exception, and you want to intercept that error and act appropriately.

Example: A user-controlled system that reports the value of a priority queue, and the queue itself is empty
EXCEPTIONS - SYNTAX

Raising

```java
public class PriorityQueue<K,V>{
    private V _min;

    // other methods elided
    // assumes somewhere we have a definition of EmptyQueueException

    public V min(){
        if (this.isEmpty()) {
            throw new EmptyQueueException("Can’t get the min of an empty PQ!");
        }
        return _min;
    }
}

No need for else case here, since exceptions stop the running of the method.
```
EXCEPTIONS – SYNTAX (2)

**Catching**

```java
public class QueueReporter {
    private PriorityQueue<int, String> _pq;

    // other methods elided

    public void reportMin(){
        try {
            System.out.println("The current min is "+_pq.min());
        } catch(EmptyQueueException e) {
            System.out.println("The queue is currently empty.");
        }
    }
}
```

- If you care about the details of the exception beyond type, that info is contained in the variable `e`
- This is just an example to show the syntax. In this case it might just be smarter to check the size of `pq`
EXCEPTIONS - NOTES

You can have multiple catch blocks to handle different exceptions.

- Syntax is what you might expect:

```java
public void someMethod()
{
    try {
        // some code
    }
    catch(ExceptionType1 e1) {
        System.out.println("12345");
    }
    catch(ExceptionType2 e2) {
        System.out.println("ABCDE");
    }
}
```

This code prints 12345 if the code in the try block throws an error of type ExceptionType1 and ABCDE if the exception is of type ExceptionType2.
Since all exceptions are subclasses of Exception, putting Exception as the type in the catch block will catch any exception

- In general this is not a good idea. You generally don’t want to just accept that something failed and sweep it under the rug, you want to do something with the error!

```java
public void verySensitiveMethod(){
    Person recipient = this.getNemesis(); // just a placeholder, going to change
    try {
        this.replaceWithActualRecipient(recipient); // can fail to reset
    }
    catch(Exception e) {
        System.out.println("Uh oh!");
    }
    this.sendLotsOfMoney(recipient);
}
```

- If you know the type of exception, you may know whether it is okay to proceed with sending the money
EXCEPTIONS – NOTES (3)

• In addition to try and catch, there is also a very important third kind of block, the finally block
  • Just like the name says, it should happen after everything else.
  • In fact, it happens whether or not the exception was caught. It always happens when a try block exits

```java
public void fileIOMethod() {
    try {
        // error-prone file operations
    }
    catch(Exception e) {
        // notify user that something went wrong
    }
    finally {
        // close up files and other resources
    }
}
```

• This way, the resources are closed up even if the exception is not an IOException
• However, if it is an IOException, perhaps we can handle it better
JUNIT TESTING!

• The reason for JUnit testing is to test individual parts of your program, ensuring that each component functions correctly.

• This is extremely useful, because as the size of projects grows – the more impact it has – the more time needs to be spent testing, as even small failures can be problematic!

• We’ve provided you with a JUnit Test file with some example tests. You are required to add you own tests in order to fully cover your code!

• It is convention to name the test file <name of class>Test.java
JUNIT TESTING (2)

```java
public class MyHeapTest {

    /**
     * A simple test to ensure that insert() works.
     */
    @Test
    public void testInsertOneElement() {
        // set-up
        MyHeap<Integer, String> heap = new MyHeap<Integer, String>(new IntegerComparator());
        heap.insert(1, "A");

        // Assert that your data structure is consistent using
        // assertThat(actual, is(expected))
        assertEquals(heap.size(), 1);
        assertEquals(heap.min().getKey(), 1);
    }
}
```
Use this format to test throwing exceptions!

```java
@Test(expected = EmptyTreeException.class)
public void testRemoveThrowsEmptyTreeException() {
    MyLinkedHeapTree<Integer> tree = new MyLinkedHeapTree<Integer>();
    tree.remove();
}
```
CODING INCREMENTALLY?

While coding Heap, it is important to keep in mind the dependencies you will need to get things working. Here is one way you could think about approaching this project incrementally:

- Start with MyHeapEntry or MyLinkedHeapTree.
  - For testing MyLinkedHeapTree, you will need to make sure that all of MyHeapEntry’s methods have been filled in.
- Then, MyHeap (this needs MyLinkedHeapTree as the underlying data structure)

Take advantage of JUnit testing to help you code incrementally! Once you get MyLinkedHeapTree, write tests for say, the add() method using your already written MyHeapEntry as the Position parameter. From the tests, you can know what you require the add() method to be doing, then you can go and write that up and make sure your tests pass it.
YOU CAN DO IT!

(BUILD SOMETHING OUT OF THIS WORLD!)