Lecture 13

Generics, part 2

Consider this code

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

- Does it compile?
- Does it run?

Consider this code

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

- Does it compile?
  - Yes.
- Does it run?
  - No! ClassCastException

How about this?

```
List<Integer> intList = new LinkedList<>();
intList.add(new Integer(0));
...
String s = (String) intList.get(0);
```

- Does it compile?

How about this?

```
List<Integer> intList = new LinkedList<>();
intList.add(new Integer(0));
...
String s = (String) intList.get(0);
```

- Does it compile?
  - No. Generics performs a stronger type checking at compile time.

What does this snippet print?

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

What does this snippet print?

This returns type 'Class' not a string.

A. True

B. False

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?

This returns type 'Class' not a string. In this case, returns class ArrayList.

A. True

Behavior is same for all classes. Hence the name 'generic'.

B. False

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

Java Generics - Erasure

```
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;
       }
}
```

Generics and subclasses

Assume you have a class Dog that extends a class Animal. Is the following legal?

```
Animal myPet = new Dog("Fido");
```

A. Yes
B. No

Generics and subclasses

Assume you have a class Dog that extends a class Animal. Is the following legal?

```
LinkedList<Animal> animalList;
LinkedList<Dog> dogList = new LinkedList<Dog>();
animalList = dogList;
```

A. Yes
B. No

Non-covariance of Generics

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

- A: 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

```
10  // Display all Shape objects in the given Collection.
11  // Call their display() instance method to do that.
12  static void displayShapes( Collection<Shape> collection ){
13      for ( Shape shape : collection )
14          shape.display() ;
15  }
 .  .  .
26  Collection<Shape> shapes = new LinkedList<Shape>();
27  shapes.add( new Circle( 5.0 ) );
28  shapes.add( new Rectangle( 4.5, 21.2 ) );
29  displayShapes( shapes );
30
31
32
33  Collection<Circle> circles = new LinkedList<Circle>();
34  circles.add( new Circle( 5.0 ) );
35  circles.add( new Circle( 15.0 ) );
36  circles.add( new Circle( 25.0 ) );
37  displayShapes( circles );          // ERROR!
```

GENERICS AND WILD CARDS

Problem

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

```
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.

```
static void displayShapes(Collection<?> 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 **any** type.

- What we want is "**any** type that **extends** Shape"

```
<? extends Shape>
```

Accept any type that is 'upper bounded' by Shape

Bounded wildcards

```
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) {
```

A. Collections<E>

B. Collections<?>

C. Collection<? extends E>

D. Collection<? super E>

E. More than one of these will work

Hw6 starter code line

- `Class myHeap<T extends Comparable<? Super T>>…`

Constraints on T

- Sometimes the type T can't be "just any old Object",

Type T must sometimes satisfy *some conditions*.

- An example of this is the **dHeap** class you are going to implement is your PA 6.

- The elements must all be **Comparable** -- the heap implementation needs to be able to call **compareTo (o)** on every element stored in the tree.

Constraints on T

- Suppose we add three objects to a heap:

```
heap = new dHeap<Object>();
heap.add("Marina"); //OK, String is Comparable
heap.add("Robert"); //OK, String is Comparable
heap.add(new Object()); //Not OK, Object not Comparable
```

Constraints on T

- Suppose we add three objects to a heap:

```
heap = new dHeap<Object>();
heap.add("Marina"); //OK, String is Comparable
heap.add("Robert"); //OK, String is Comparable
heap.add(new Object()); //Not OK, Object not Comparable

if (node[index1].compareTo(node[index2]<0) {…
```

But if index1 refers to the Object we added, this method will fail because Object does not implement the `Comparable` interface .

Comparable Interface

Methods

Modifier and Type	Method and Description
`int`	**`compareTo`**`(T o)` Compares this object with the specified object for order.

Method Detail

compareTo

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Bounds on type parameters

- What we want is a way of enforcing that the type parameter T be of type `Comparable`.

- Bounds on type parameters.

Bounds on type parameters

- We can also require that type T *implement* some interface.
- For example, `dHeap` class should only store elements that are all `Comparable`.

```
class dHeap <T extends Comparable> implements dHeapIntfce<T>
```

- The "extends Comparable" enforces that any T we pass in as the type parameter must be of type `Comparable`.

dHeap<Object> heap = new dHeap <Object>() // not ok. Compile Error

Bounds on type parameters

- In the previous example, `Comparable` was the upper bound of `T`.
- The `Comparable` interface takes a type parameter of its own.

```
interface Comparable<U> {
  int compareTo (U o);
}
```

(In the previous example, we used the `Comparable` interface in "compatibility mode", where we did not specify `U`).

- The type parameter `U` specifies what kinds of objects `o` we should be able to compare to.

Bounds on type parameters

- We can define what kind of objects U we can compareTo:

- Example:

```
class dHeap<T extends Comparable<T>>…
```

- Here we require that whatever type T the `dHeap` is instantiated with, it must be `Comparable` to other objects of type T

Bounds on type parameters

- Consider the following example:

```
class B { }
class A implements Comparable<B> {
  int compareTo (B o) {
    return 0;
  }
}
```

- Given the definitions above, an object of type A can *only* be compared to objects of type **B**.

```
final A a = new A();
final B b = new B();
final int result = a.compareTo(b);   // OK
```

  - We *cannot* compare **a** to another object of type **A**!

Bounds on type parameters

- Given

```
class dHeap<T extends Comparable<T>>…
```

If we try to instantiate a dHeap with A as the type parameter

```
dHeap<A> heap = new dHeap<A>;   //not OK!
```

- This error occurs because, even though **A** is **Comparable** to *something* (B), it is not **Comparable<A>.**

Bounds on type parameters

- On the other hand,
  - **String** implements **Comparable<String>**
  - **Integer** implements **Comparable<Integer>**
- Both String and Integer would be accepted as type parameters for dHeap:

- `dHeap<String> heap = new dHeap<String>;`
- `dHeap<Integer> heap = new dHeap<Integer>;`
- //both are fine

Bounds on type parameters

- While useful, our current definition of `dHeap` is a bit overly restrictive:

- Consider a hierarchy of **Shape** classes:

```
class Shape implements Comparable<Shape> {
   int compareTo (Shape o) { ... }
}
class Rectangle extends Shape {
  ...
}
```

- The **Rectangle** class inherits the **compareTo (Shape o)** method from its parent **Shape** class.

Bounds on type parameters

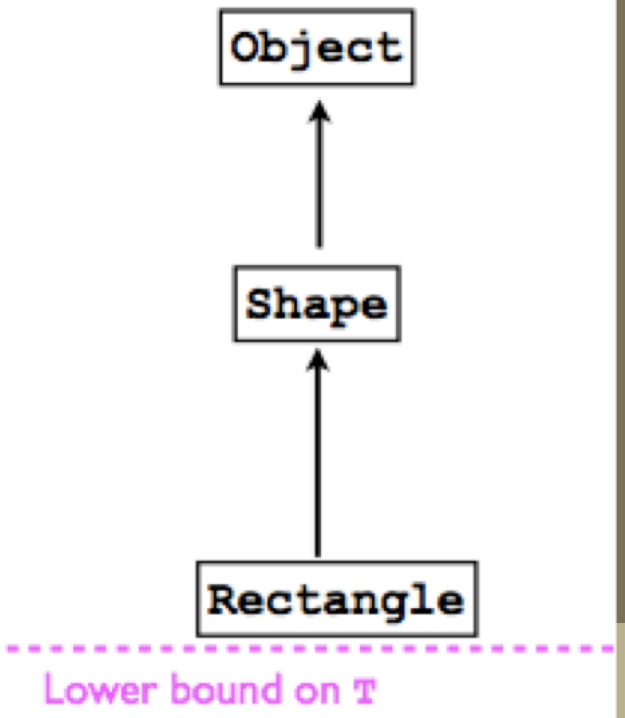
- However, **Rectangle** does not offer a method **compareTo (Rectangle o)** designed specifically for other **Rectangle** objects.

- Hence, the `Rectangle` class could not be used as the type parameter T when instantiating a dHeap:

- *Reason*: Even though **`Rectangle`** is **`Comparable`** to other **`Shape`** objects, it is not **`Comparable<Rectangle>`**.
  - I.e., **`Rectangle`** offers no **`int compareTo (Rectangle o)`** method.

Lower bounds on types

- What we need is a way of expressing that type parameter `T` may be `Comparable` with class `T`, *or any super-class of* `T`.
  - E.g., we want to allow dHeap to store `Rectangle` objects:
    - `Rectangle`s are all `Comparable` with `Shape`, where `Shape` is a *super-class* of `Rectangle`.
- To solve this problem, Java offers **lower bounds** on type parameters.



Lower bounds on types

- For example, we can allow the `dHeap` class to accept any type T as long as T is `Comparable` to class T OR any super-class of T.

- `Class dHeap<T extends Comparable<? Super T>>…`

- The wildcard type ? Indicates:
  - "We don't care which type T is Comparable to, so long as it's Comparable to some super-class of T (or T itself)."
  - The keyword super indicates the lower bound of the type parameter.

Lower bounds on types

- Given this revised definition of dHeap, we can now instantiate a heap of Rectangle objects:

- dHeap<Rectangle> heap = new dHeap<Rectangle>(); // OK

Generics... can you do:

- Which of the following compile?

```
1. Collection<List> c = new LinkedList<List>();
2. LinkedList<List> myL = new LinkedList<List>();
3. LinkedList<List> myL = new LinkedList<ArrayList>();
4. LinkedList<? extends List> myL = new LinkedList<ArrayList>();
5. LinkedList<? super List> myL = new LinkedList<List>();
6. LinkedList<? super List> myL = new LinkedList<Collection>();
7. LinkedList<Collection> myL = new LinkedList<Collection>();
8. myL.add( new ArrayList() );
```