Northeastern University

Lecture 24: More about Generics

Prof. Chen-Hsiang (Jones) Yu, Ph.D.

College of Engineering

Materials are edited by Prof. Jones Yu from

Data Structures and Abstractions with Java, 5th edition. By Frank M. Carrano and Timothy M. Henry.
ISBN-13 978-0-13-483169-5 © 2019 Pearson Education, Inc.

The Interface `Comparable`

- Consider the method `compareTo` for class `String`
- If s and t are strings, `s.compareTo(t)` is
  - » Negative if s comes before t
  - » Zero if s and t are equal
  - » Positive if s comes after t

The Interface `Comparable`

- By invoking `compareTo`, you compare two objects of the class T.
- The interface `java.lang.Comparable`

```
public interface Comparable<T>
{
    public int compareTo(T other);
} // end Comparable
```

The Interface `Comparable`

- Create a class `Circle`, define `compareTo`

```
public class Circle implements Comparable<Circle>, Measurable
{

   private double radius;

   // Definitions of constructors and methods are here.
   // . . .

   public int compareTo(Circle other)
   {
      int result;
      if (this.equals(other))
         result = 0;
      else if (radius < other.radius)
         result = -1;
      else
         result = 1;

      return result;
   } // compareTo

} // end Circle
```

Generic Methods

- Suppose you have a class that does not define a type parameter in its header, but you want to use a generic data type in a method of that class.
- Take following steps:
  - Write a type parameter enclosed in angle brackets in the method's header just before its return type
  - Use the type parameter within the method as you would if it were in a generic class

Generic Methods

Different class types

```
public class Example
{
    public static void main(String args[])
    {
       String[] stringArray = {"apple", "banana", "carrot", "dandelion"};
       System.out.print("stringArray contains ");
       displayArray(stringArray);

       Character[] characterArray = {'a', 'b', 'c', 'd'};
       System.out.print("characterArray contains ");
       displayArray(characterArray);
    } // end main

    public static <T> void displayArray(T[] anArray)
    {
       for (T arrayEntry : anArray)
       {
          System.out.print(arrayEntry);
          System.out.print(' ');
       } // end for
       System.out.println();
    } // end displayArray

} // end Example
```

Bounded Type Parameters

- Consider this simple class of squares:

```
public class Square<T>
{
   private T side;

   public Square(T initialSide)
   {
      side = initialSide;
   } // end constructor

   public T getSide()
   {
      return side;
   } // end getSide
} // end Square
```

- Different types of square objects are possible

```
Square<Integer> intSquare = new Square<>(5);
Square<Double> realSquare = new Square<>(2.1);
Square<String> stringSquare= new Square<>("25");
```

Exercise (L24_E1)

- Please type following code to your Eclipse and fix the error.

```
public class Square<T>
{
   private T side;

   public Square(T initialSide)
   {
      side = initialSide;
   } // end constructor

   public T getSide()
   {
      return side;
   } // end getSide

   public double getArea() {
       double s = side.doubleValue();
       return s*s;
   }
} // end Square
```

Answer

- We want the `side` of a square to be a numeric value.
- We can impose this restriction by making T represent a class that is derived from `Number`, the base class (superclass) of the classes Byte, Double, Float, Integer, Long and Short.
- We bound T by using "`T extends Number`" in `Square's` header.

Answer

```
public class Square<T extends Number>
{
   private T side;

   public Square(T initialSide)
   {
      side = initialSide;
   } // end constructor

   public T getSide()
   {
      return side;
   } // end getSide

   public double getArea() {
       double s = side.doubleValue();
       return s*s;
   }
} // end Square
```

Bounded Type Parameters (cont.)

- Imagine that we want to write a static method that returns the smallest object in an array. Suppose that we wrote our method shown here:

```
public MyClass
{
   // First draft and INCORRECT:
   public static <T> T arrayMinimum(T[] anArray)
   {
      T minimum = anArray[0];
      for (T arrayEntry : anArray)
      {
         if (arrayEntry.compareTo(minimum) < 0)
            minimum = arrayEntry;
      } // end for

      return minimum;
   } // end arrayMinimum
} // end MyClass
```

Bounded Type Parameters

- Header really should be as shown

```
public MyClass
{
   public static <T extends Comparable<T>> T arrayMinimum(T[] anArray)
   {
      T minimum = anArray[0];
      for (T arrayEntry : anArray)
      {
         if (arrayEntry.compareTo(minimum) < 0)
            minimum = arrayEntry;
      } // end for

      return minimum;
   } // end arrayMinimum

   // . . .

} // end MyClass
```

Exercise

- What, if anything, is wrong with the following class?

```
public final class Min {
   public static T smallerOf(T x, T y) {
     if (x < y)
      return x;
     else
      return y;
   }
}
```

Answer

```
public final class Min {
   public static <T extends Comparable<T>> T smallerOf(T x, T y) {
     if (x.compareTo(y) <0)
      return x;
     else
      return y;
   }
}
```

Wildcards

- Question mark, ?, is used to represent an unknown class type
  - » Referred to as a wildcard

- Consider following method and objects

```
public static void displayPair(OrderedPair<?> pair)
{
   System.out.println(pair);
} // end displayPair
```

Wildcards

- Method `displayPair` will accept an argument, a pair of objects, whose data type is any class.

```
OrderedPair<String> aPair = new OrderedPair<>("apple", "banana");
OrderedPair<Integer> anotherPair = new OrderedPair<>(1, 2);

displayPair(aPair);
displayPair(anotherPair);
```

Bounded Wildcards

- Recall the method `arrayMinimum()`

```
public MyClass
{

   public static <T extends Comparable<T>> T arrayMinimum(T[] anArray)
   {
      T minimum = anArray[0];
      for (T arrayEntry : anArray)
      {
         if (arrayEntry.compareTo(minimum) < 0)
            minimum = arrayEntry;
      } // end for

      return minimum;
   } // end arrayMinimum

   // . . .

} // end MyClass
```

Bounded Wildcards (cont.)

- Then, we called this method with the statement

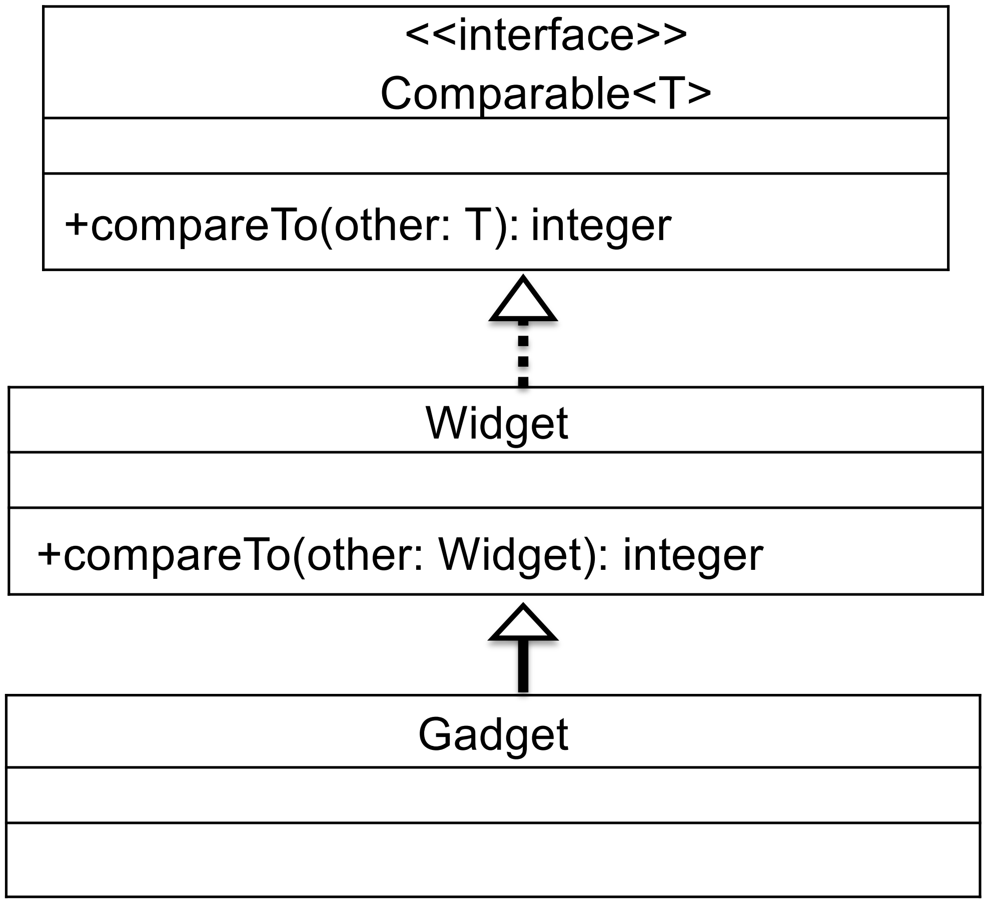
```
Gadget smallestGadget = MyClass.arrayMinimum(myArray);
```

where `myArray` is an array of `Gadget` objects.

- Instead of comparing an object of T only with other objects of T, we can allow comparisons to objects of a superclass of T.

`<T extends Comparable<T>>` ➡ `<T extends Comparable<? super T>>`

Bounded Wildcards (cont.)


<<interface>>
Comparable<T>
+compareTo(other: T): integer
Widget
+compareTo(other: Widget): integer
Gadget

The class `Gadget` is derived from the class `Widget`, which implements the interface `Comparable`

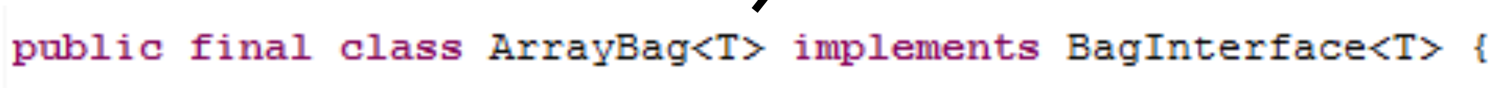
More About Generics

```
public static <T extends Comparable<? super T>>
        void selectionSort(T[] a, int n)
{
```

More About Generics (cont.)

- Generic Methods
  - If you have a class that does not define a type parameter in its header

define a type parameter



```
public final class ArrayBag<T> implements BagInterface<T> {
```

  - But, you want to use a generic data type in a method of that class.

More About Generics (cont.)

```
<T extends Comparable<? super T>>
```

- T represents a class that derived from Comparable, i.e., we bound T by writing `T extends Comparable`.

```
<? super T>
```

- ? : represent an unknown class type and is referred to as a wildcard.
- `? super T` means any superclass of T

More About Generics (cont.)

- Q: Why not use “implements” instead of “extends” in

```
<T extends Comparable<? super T>>
```

- As the official document mentions:

To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound, ... Note that, in this context, extends is used in a general sense to mean either "extends" (as in classes) or "implements" (as in interfaces).

https://docs.oracle.com/javase/tutorial/java/generics/bounded.html