Introduction to Generics/Sorting

CST 365 – Computer Science II
Michael Ruth, Ph.D.
Assistant Professor
Computer Science & I.T.
mruth@roosevelt.edu

Objectives

- Introduce the general generics concept and its role in programming languages
- Discuss the use of generics in Java
- Introduce Collections convenience class
- Discuss natural and imposed order and the mechanisms used to do either
- Explain the use of the Comparable interface to create a natural order
- Discuss the use of the Comparator interface to create a imposed order

Generic Classes and Methods

- A ***generic*** class or method is one whose definition uses a placeholder for one or more of the types it works with
  - The placeholder is really a type parameter
- For a generic class
  - the type argument is specified when an object of the generic class is being instantiated
- For a generic method
  - the compiler deduces the type argument from the type of data being passed to the method

The ArrayList Class

- The **`ArrayList`** class is generic:
  - the definition of the class uses a type parameter for the type of the elements that will be stored
- Examples:
  - **`ArrayList<String>`** specifies a version of the generic **`ArrayList`** class that can hold **`String`** elements only
  - **`ArrayList<Integer>`** specifies a version of the generic **`ArrayList`** class that can hold **`Integer`** elements only.

Instantiation and Use

- **`ArrayList<String>`** is used as if it was the name of any non-generic class:

```
ArrayList<String> myList = new ArrayList<String>();
myList.add("Java is fun");
String str = myList.get(0);
```


A Generic Point Class

- Consider a "point" as a pair of coordinates x and y, where x and y may be of any one type
  - IE, X & Y are the same type

Point Example (1)

```
class Point<T> {

  private T x, y;

  public Point(T x, T y) {
     set(x, y);
  }

  public void set(T x, T y)
  {
      this.x = x;   this.y = y;
  }

  T getX(){ return x;}

  T getY(){ return y;}

}
```


Point Example (2)

```
public class Test
{
    public static void main(String [] s)
    {
       Point<String> strPoint =
            new Point<String>("Anna", "Banana");
       System.out.println(strPoint);
       Point<Number> pie =
            new Point<Number>(3.14, 2.71);
       System.out.println(pie);
    }
}
```


Reference Types and Generic Class Instantiation

- Only reference types can be used to declare or instantiate a generic class:

```
ArrayList<Integer> myIntList = new ArrayList<Integer>;
ArrayList<int> myIntList = new ArrayList<int>;
```

- int is not a reference type, so it cannot be used to declare or instantiate a generic class
  - You must use the corresponding wrapper class
    - IE, Integer for int, Double for double, etc.

Not Specifying a Type

- You can create an instance of a generic class without specifying the actual type argument
  - An object created in this manner is said to be of a raw type (really, it's an object ☺)
    - `Point rawPoint = new Point("Anna", new Integer(26));`
  - It is necessary for the programmer to keep track of types used and use casting:
    - `String name = (String)rawPoint.getX();`

Commonly Used Type Parameters

Name	Usual Meaning
T	Used for a generic type.
S	Used for a generic type.
E	Used to represent generic type of an element in a collection.
K	Used to represent generic type of a key for a collection that maintains key/value pairs.
V	Used to represent generic type of a value for collection that maintains key/value pairs.

Generic Parameters

- Consider a method that returns the square length of a **`Point`** object with numeric coordinates

```
static int sqLength(Point<Integer> p){
   int x = p.getX();
   int y = p.getY();
   return x*x + y*y;
}
```

Wildcard Parameters

- Generic type checking is very strict, so we can use a wildcard type symbol **?** stands for any generic type
  - These wildcards can be constrained to subclasses of specified classes!

Across An Entire Class

```
class Point<T extends Number> {

  private T x, y;

  public Point(T x, T y){
     this.x = x;  this.y = y;
  }

  T getX(){ return x;}

  T getY(){ return y;}

}
```


Constraining To Interfaces

- A type parameter can be constrained to a type implementing an interface:
  - `public static <T extends Comparable<T>> T greatest(T arg1, T arg2)`
  - We can call this method like:

```
Employee bigShot = new
   Employee("Joe Manager", 10);
Employee littleShot = new
   Employee("Homer Simpson", 1);

Employee greatest = greatest(bigShot, littleShot);
```

Sorting…

- The **Collections** class provide some convenience methods for the JCF
- Useful Methods like: Min, Max, Reverse, sort, shuffle, binarySearch… but…
  - We are missing some key details on ordering… how is order in the JCF controlled???

Order

- Natural Order
  - The objects provide their own ordering…
    - These would implement **`Comparable<T>`** which defines a total ordering on them
- Imposed Order
  - An order is imposed on the collection
    - Need to define an order to impose using the interface **`Comparator<T>`**

`Comparable` Returns…

- The **`compareTo`** method:
  - returns a negative integer if this object is "less than" the given object
  - returns 0 if this object is "equal" to the given object
  - returns a positive integer if this object is "greater than" the given object
- Note: String implements Comparable!

Implementing **Comparable**

```
class Employee implements Comparable<Employee> {

   private int rank;
   private String name;

   public int compareTo(Employee other)
   {
       return this.rank - other.rank;
   }

   public Employee(String n, int r)
   {
       rank = r; name = n;
   }

   public String toString()
   {
       return name + " : " + rank;
   }
}
```


The **Comparator** Interface

- Comparator is generic as well and has two methods (we usually ignore one)
  - **int compare(Object o1, Object o2)**
    - returns negative int if o1 is "less than" o2
    - returns 0 if o1 is "equal" to o2
    - returns positive int if o1 is "greater than" o2

Implementing Comparator

```
class EmpCompareSals implements Comparator<Employee> {

   public int compare(Employee one, Employee two)
   {
       return one.getSalary() - two.getSalary();
   }

}
```

`Collections` methods

- Sorting:
  - `public static <T extends Comparable<? super T>> void sort(List<T> list)`
  - `public static <T> void sort(List<T> list, Comparator<? super T> c)`
- Searching (Note: BS requires sorted list):
  - `public <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)`
  - `public <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)`

Summary

- Introduced the general generics concept and its role in programming languages
- Discussed the use of generics in Java
- Introduced Collections convenience class
- Discussed natural and imposed order and the mechanisms used to do either
- Explained the use of the Comparable interface to create a natural order
- Discussed the use of the Comparator interface to create a imposed order

Questions?