

CMSC 202

Generics II

Generic Sorting

We can now implement sorting functions that can be used for any class (that implements Comparable). The familiar insertion sort is shown below.

```
public static <T extends Comparable<T>> void insertionSort(T[] array) {  
    for (int i = 1; i < array.length; i++) {  
        int j = i;  
        T item = array[i];  
        while ((j > 0) && (array[j - 1].compareTo(item) > 0)) {  
            array[j] = array[j - 1];  
            j--;  
        }  
        array[j] = item;  
    }  
}
```

Generics and Hierarchies

- What if we want a somewhat specialized container that assumes the objects it holds are part of a hierarchy so that the container code can assume the existence of a particular method?
- Let's look at animals, dogs, and cats.

```
class Animal { public void speak(){...} ... }  
class Dog extends Animal {...}  
class Cat extends Animal {...}
```

We would like to create a container named Zoo to hold some animals that speak.

Zoo<T>

- If we define the Zoo like this

```
public class Zoo< T >
```

we'll get a compiler error where the speak() method is called. Not all classes provide a method called speak(). Only Animal and classes that extend it provide a speak().

The solution is to place a bounds on T. The Zoo can only contain Animal references or any type that inherits from Animal.

```
public class Zoo< T extends Animal >
```

The phrase **T extends Animal** means “Animal or any subtype of Animal.”

Generics and Hierarchy

For example, suppose we revisit the Animals.

Each animal has a weight and a name. Let's say two Dogs (or two Cats) are equal if they have the same name and weight.

```
class Animal { private String name; private int weight; ...}  
class Dog extends Animal implements Comparable<Dog>{ ... }  
class Cat extends Animal implements Comparable<Cat>{ ... }
```

Since Dog implements Comparable<Dog>, it's clear you can compare Dogs with Dogs, but not with Cats.

So, we can use our insertionSort() method with Dogs or with Cats.

Sorting Dogs

```
public void main(String[ ] args) {  
    Dog[ ] dogs = new Dog[42];  
    // Put some dogs in the array  
    . . .  
    // Use insertion sort to sort dogs  
    MyClass.insertionSort(dogs);  
}
```

Generics and Hierarchies

What happens if we want to sort a class in an inheritance hierarchy, and some ancestor of the class implements Comparable, but not the class itself?

But suppose we wanted to compare all Animals using only their weight? The class definitions would look something like this.

```
class Animal implements Comparable<Animal> { ... }  
class Dog extends Animal { ... }  
class Cat extends Animal { ... }
```

Since Animal implements comparable, any two Animals can be compared (albeit only by weight).

The problem is now that we can't use insertionSort to sort an array of Dogs because Dog doesn't explicitly implement Comparable (it's inherited from Animal).

New insertionSort

The solution is to use a “wildcard” when defining insertionSort.

? **super T** is read as “T or any supertype of T”. Now, because Dog extends Animal which implements Comparable, insertionSort can be used with an array of Dogs as before.

```
public static <T extends Comparable<? super T>> void insertionSort(T[] array) {  
    for (int i = 1; i < array.length; i++){  
        int j = i;  
        T item = array[i];  
        while ((j > 0) && (array[j-1].compareTo(item) > 0)){  
            array[j] = array[j-1];  
            j--;  
        }  
        array[j] = item;  
    }  
}
```


Pitfall: A Generic Class Cannot Be an Exception Class

- It is not permitted to create a generic class with **Exception**, **Error**, **Throwable**, or any descendent class of **Throwable**.
 - A generic class cannot be created whose objects are throwable.

```
public class GEx<T> extends Exception
```

- The above example will generate a compiler error.

Tip: Generic Interfaces

- An interface can have one or more type parameters.
- The details and notation are the same as they are for classes with type parameters.

Generic Methods

- When a generic class is defined, the type parameter can be used in the definitions of the methods for that generic class.
- In addition, a generic method can be defined that has its own type parameter that is not the type parameter of any class.
 - A generic method can be a member of an ordinary class or a member of a generic class that has some other type parameter.
 - The type parameter of a generic method is local to that method, not to the class.

Generic Methods

- The type parameter must be placed (in angle brackets) after all the modifiers, and before the return type.

```
public class Utility {  
    ...  
    public static <T> T getMidPoint(T[] array) {  
        return array[array.length / 2];  
    }  
  
    public static <T> T getFirst(T[] a) {  
        return a[0];  
    }  
    ...  
}
```

Generic Methods

- When one of these generic methods is invoked, the method name is prefaced with the type to be plugged in, enclosed in angular brackets.

```
String s =  
    Utility.<String>getMidPoint(arrayOfStrings);
```

```
double first =  
    Utility.<Double>getFirst(arrayOfDoubles);
```

Inheritance with Generic Classes

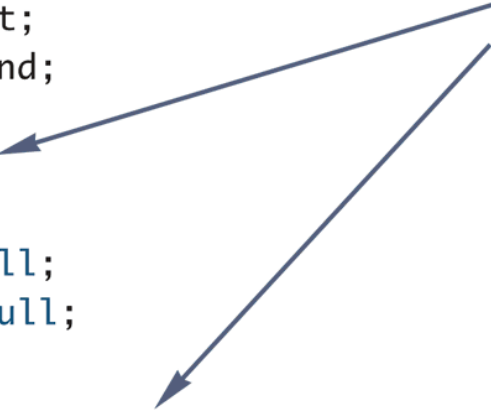
- A generic class can be defined as a derived class of an ordinary class or of another generic class.
 - As in ordinary classes, an object of the subclass type would also be of the superclass type.
- Given two classes **A** and **B**, and a generic class **G**, there is no relationship between **G<A>** and **G**.
 - This is true regardless of the relationship between class **A** and **B**, e.g. if class **B** is a subclass of class **A**.

The Commonly Used Generic Ordered Pair Class (1 of 4)

Display 14.5 A Generic Ordered Pair Class

```
1 public class Pair<T>
2 {
3     private T first;
4     private T second;
5     public Pair()
6     {
7         first = null;
8         second = null;
9     }
10    public Pair(T firstItem, T secondItem)
11    {
12        first = firstItem;
13        second = secondItem;
14    }
```

Constructor headings do not include the type parameter in angular brackets.



(continued)

A Generic Ordered Pair Class (2)

Display 14.5 A Generic Ordered Pair Class

```
15     public void setFirst(T newFirst)
16     {
17         first = newFirst;
18     }

19     public void setSecond(T newSecond)
20     {
21         second = newSecond;
22     }

23     public T getFirst()
24     {
25         return first;
26     }
```

(continued)

A Generic Ordered Pair Class (3)

Display 14.5 A Generic Ordered Pair Class

```
27     public T getSecond()
28     {
29         return second;
30     }

31     public String toString()
32     {
33         return ( "first: " + first.toString() + "\n"
34                 + "second: " + second.toString() );
35     }
36
```

(continued)

A Generic Ordered Pair Class (4)

Display 14.5 A Generic Ordered Pair Class

```
37     public boolean equals(Object otherObject)
38     {
39         if (otherObject == null)
40             return false;
41         else if (getClass() != otherObject.getClass())
42             return false;
43         else
44         {
45             Pair<T> otherPair = (Pair<T>)otherObject;
46             return (first.equals(otherPair.first)
47                 && second.equals(otherPair.second));
48         }
49     }
50 }
```

A Derived Generic Class (1 of 2)

In this example, `UnorderedPair` overrides the `equals()` method that was inherited from `Pair`.

Display 14.11 A Derived Generic Class

```
1 public class UnorderedPair<T> extends Pair<T>
2 {
3     public UnorderedPair()
4     {
5         setFirst(null);
6         setSecond(null);
7     }
8
9     public UnorderedPair(T firstItem, T secondItem)
10    {
11        setFirst(firstItem);
12        setSecond(secondItem);
13    }
```

(continued)

A Derived Generic Class (2)

Display 14.11 A Derived Generic Class

```
13     public boolean equals(Object otherObject)
14     {
15         if (otherObject == null)
16             return false;
17         else if (getClass() != otherObject.getClass())
18             return false;
19         else
20         {
21             UnorderedPair<T> otherPair =
22                 (UnorderedPair<T>)otherObject;
23             return (getFirst().equals(otherPair.getFirst())
24                 && getSecond().equals(otherPair.getSecond()))
25                 ||
26                 (getFirst().equals(otherPair.getSecond())
27                 && getSecond().equals(otherPair.getFirst()));
28         }
29     }
30 }
```

Using UnorderedPair (1 of 2)

Display 14.12 Using UnorderedPair

```
1 public class UnorderedPairDemo
2 {
3     public static void main(String[] args)
4     {
5         UnorderedPair<String> p1 =
6             new UnorderedPair<String>("peanuts", "beer");
7         UnorderedPair<String> p2 =
8             new UnorderedPair<String>("beer", "peanuts");
```

(continued)

Using UnorderedPair (2)

Display 14.12 Using UnorderedPair

```
9         if (p1.equals(p2))
10        {
11            System.out.println(p1.getFirst() + " and " +
12                               p1.getSecond() + " is the same as");
13            System.out.println(p2.getFirst() + " and "
14                               + p2.getSecond());
15        }
16    }
17 }
```

SAMPLE DIALOGUE²

```
peanuts and beer is the same as
beer and peanuts
```