

Generic Methods and Classes

(CISS-111 William Jojo)

Overview

Much of the code we have dealt with up to this point has been geared toward a certain data type. For example, a certain method or even a class has been centered around strings or integers on many occasions.

Now this may be useful for some instances, but there comes a time when we have to write a piece of code *generically*. That is, we need the method or class to accept *any* data type we pass to it and still perform the actions accurately.

Both methods and classes support generic data type handling. We will start our discussion with two interfaces `Cloneable` and `Comparable` since these interfaces allow us to override some standard methods of the `Object` class. By doing so, we gain better support in handling data generically.

The `Cloneable` Interface

Recall from our discussion of arrays that they can be cloned using the `clone()` method. This is because the array class implements the `Cloneable` interface. The `clone` method is a protected method of the `Object` superclass. Recall that all objects are descendants of the `Object` superclass. This method is protected and does not work for just any class we create or existing in some package. Instead we must elect to implement the interface and override the `clone()` method with a public one.

Any attempt to use the `clone()` method with implementing the interface and overriding the `clone()` method with a public one will result in the following error:

```
clone() has protected access in java.lang.Object
```

Example 1 shows a basic implementation of cloning with the important details in bold.

```

public class CloneEx1 {

    public static class MyClass implements Cloneable {
        String s;    // immutable String
        int a, b;    // primitive types

        public MyClass (String s, int a, int b) {
            this.s = s;
            this.a = a;
            this.b = b;
        }

        public Object clone() {
            MyClass dup;

            try {
                dup = (MyClass) super.clone();
            }
            catch (CloneNotSupportedException e) {
                return null;
            }

            return dup;
        }
    }

    public static void main(String[] args) {
        MyClass orig, copy;

        orig = new MyClass("Computer", 6, 24);
        copy = (MyClass) orig.clone();
    }
}

```

Example 1: Program to demonstrate the use of clone() method.

The act of cloning returns a field-for-field copy of the data in the object. The intent of the copy is to be a complete duplicate of the information. It is important for us to recognize the difference between a clone and simply using the statement:

```
copy = orig;
```

That statement only make copy point to the same object whereas we wanted an exact

duplicate of the object.

Since there exists only one `String` object with the value "Computer" we do not have to worry about an explicit clone of the instance variable `s`. What if it was not a string but another object like an array? Consider the code in *Example 2*.

```
public class CloneEx2 {  
  
    public static class MyClass implements Cloneable {  
        String s;    // immutable String  
        int ia[];    // array class object  
  
        public MyClass (String s, int len) {  
            this.s = s;  
            ia = new int[len];  
        }  
  
        public Object clone() {  
            MyClass dup;  
  
            try {  
                dup = (MyClass) super.clone();  
                dup.ia = (int[]) ia.clone();  
            }  
            catch (CloneNotSupportedException e) {  
                return null;  
            }  
  
            return dup;  
        }  
    }  
  
    public static void main(String[] args) {  
        MyClass orig, copy;  
  
        orig = new MyClass("Grades", 50);  
        copy = (MyClass) orig.clone();  
    }  
}
```

Example 2: Program to demonstrate the need for deep copying when cloning an object.

If we did not explicitly clone the `ia` instance variable, we would have ended up with a copy of

the *reference* to the original array object. We must always be mindful of our objects when implementing clone otherwise we may alter data that is referenced by another object. These can be very time consuming bugs to resolve.

The Comparable Interface

This Comparable interface allows us to create a compareTo() method to properly compare two objects. Recall that the String class uses a similar method to compare strings and returns a positive, negative or zero value to indicate if a string is greater than, less than or equal to the other respectively like so:

```
String a = "abc", b = "abb";

if (a.compareTo(b) > 0)
    System.out.println("a is greater than b.");
else if (a.compareTo(b) < 0)
    System.out.println("a is less than b.");
else
    System.out.println("a and b are equal");
```

We will now consider a class that implements both interfaces Cloneable and Comparable. *Example 3* demonstrates this with a Time class that represents clock time.

Within the Time class we have stripped much of the checking that would ordinarily be in place. All of the checks for accurate time including making sure minutes and seconds are in the range of 0-59 and that 24-hour representation is in the range of 0-23 would ordinarily be present. This was done simply to focus on the two interfaces being implemented within the class.

The code in *Example 3* includes the clone() method and the new compareTo() method. We have also included an equals() method that overrides the equals() method of the Object superclass so that it accurately verifies that two times are equivalent.

```

public class Time implements Cloneable, Comparable {

    int hours, minutes, seconds;

    public Time (int hr, int min, int sec) {
        hours = hr;
        minutes = min;
        seconds = sec;
    }

    public Object clone() {
        Time dup;

        try {
            dup = (Time) super.clone();
        }
        catch (CloneNotSupportedException e) {
            return null;
        }

        return dup;
    }

    public int compareTo(Object otherTime) {
        int delta;
        Time t = (Time) otherTime;

        delta = hours - t.hours;
        if (delta != 0)
            return delta;

        delta = minutes - t.minutes;
        if ( delta != 0)
            return delta;

        return seconds - t.seconds;
    }
}

```

Example 3: Implementation of the Time class that implements both Cloneable and Comparable interfaces.

```

    public boolean equals(Object otherTime) {
        Time t = (Time) otherTime;

        return hours == t.hours && minutes == t.minutes
            && seconds == t.seconds;
    }

    public String toString() {
        return String.format("%02d:%02d:%02d", hours,
            minutes, seconds);
    }
}

```

Example 3: Time class, continued.

The code in *Example 4* exercises the `Time` class to make certain that `clone()`, `compareTo()` and `equals()` work as expected.

```

public class CompareEx {

    public static void main(String[] args) {
        Time clock1, clock2, copyof2;

        clock1 = new Time(11, 57, 0); // 11:57:00am
        clock2 = new Time(14, 0, 0); // 2:00:00pm
        copyof2 = (Time) clock2.clone();

        if (clock1.compareTo(clock2) > 0)
            System.out.println(clock1 + " is later than " + clock2);
        else if (clock1.compareTo(clock2) < 0)
            System.out.println(clock1 + " is earlier than " + clock2);

        if (clock2.equals(copyof2))
            System.out.println("clock2 and copyof2 are equal");
    }
}

```

Example 4: Program to exercise the Time class.

Generic Methods

Imagine for moment that we need to display a variable list of data. This data could be any primitive type or class. Imagine further that we have several methods to do the same work but specialized to a particular type of data. Consider the following `display()` methods:

```
public static void display(int ... items) {
    for (int x : items)
        System.out.print(x + " ");
    System.out.println();
}

public static void display(double ... items) {
    for (int x : items)
        System.out.print(x + " ");
    System.out.println();
}

public static void display(String ... items) {
    for (int x : items)
        System.out.print(x + " ");
    System.out.println();
}
```

The ellipsis (...) is used to designate that we may pass one or more items as individual, comma separated values or as an array of that type like so:

```
int x, y, z, ia[] = new int[50];

// Sometime later
display(x);
display(z);
display(x, y, z);
display(ia);
```

Although overloading allows us to create as many methods as we need to deal with all of the possible types we might want to display, it is simply inconvenient to duplicate code in this manner. Using generic methods with *type parameter* or *type variables* allows to simplify this arrangement into a single method as seen here:

```
public static <T> void display(T ... items) {
    for (T x : items)
        System.out.print(x + " ");
    System.out.println();
}
```

The bold portions of the new `display()` method indicate how the type is to be applied to the method.

First, the `<T>` is the comma delimited list of type variables that will be used in this method. This is placed just prior to specifying the return type for the method.

Second, we state that `items` is a list of *references* of type `T`. Type parameters can only represent references. This is not a problem when using primitive types since they will be auto-boxed for us in JDK5. The exception is an array of a primitive type since each element of the array is not auto-boxed when passed.

Third, the variable `x` is of type `T` to complete the foreach loop.

It is not uncommon to specify the return type, formal parameters or local variables using the type parameter. Any combination is allowed. In our `display()` example, only the return type was not specified by the type variable `T`.

It is essential to realize that we cannot instantiate objects specified by the type parameter. So although we can declare variables:

```
T refName;
```

we cannot instantiate the objects:

```
refName = new T();
```

The program in *Example 5* demonstrated using our generic `display()` method.

```

public class GenMethod {

    public static <T> void display(T ... items) {
        for (T x: items)
            System.out.print(x + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        Time clock1, clock2;
        int x=10, y=15;
        Integer[] ia={1,2,3,4,5};

        clock1 = new Time(11, 57, 0); // 11:57:00am
        clock2 = new Time(14, 0, 0); // 2:00:00pm

        System.out.print("x = ");
        display(x);

        System.out.print("y = ");
        display(y);

        System.out.print("x & y = ");
        display(x, y);

        System.out.print("ia = ");
        display(ia);

        System.out.print("clock1 & clock2 = ");
        display(clock1, clock2);
    }
}

```

Example 5: Program demonstrating a generic method with the Time class from a previous example.

We can also restrict the type of data we will process in a method by using a bounded type parameter. This is shown in the following:

```
public static <T extends Comparable<T> > T min(T a, T b) {
    if (a.compareTo(b) < 0)
        return a;
    else
        return b;
}

public static <T extends Comparable<T> > T max(T a, T b) {
    if (a.compareTo(b) >= 0)
        return a;
    else
        return b;
}
```

The expression `<T extends Comparable<T> >` simply set a different upper bound on what can be used at the type parameter. Normally this is `Object`. This states that only those objects that implement the `Comparable` interface. This is necessary because we are using the `compareTo()` method and not all classes implement that method.

Note that we are using `extends` instead of `implements`. Ordinarily we would use `implements` when specifying an interface like `Comparable`. *Whenever we are setting bounds on the type parameter we **always** use `extends`.*

The program in *Example 6* shows how we can use the new `min()` and `max()` bounded generic methods.

In this program note that the `int` and `double` type variables in the `main()` method are auto-boxed before being sent to the `min()` and `max()` methods. The output is as follows:

```
10 is smaller than 15
3.14 is smaller than 4.01
ZZZ is smaller than abd
```

```

public class Bounded {

    public static <T extends Comparable<T> > T min(T a, T b) {
        if (a.compareTo(b) < 0)
            return a;
        else
            return b;
    }

    public static <T extends Comparable<T> > T max(T a, T b) {
        if (a.compareTo(b) >= 0)
            return a;
        else
            return b;
    }

    public static void main(String[] args) {
        int a=10, b=15;
        double f=4.01, g=3.14;
        String s="ZZZ", t="abd";

        System.out.println(min(a, b) + " is smaller than " + max(a, b));
        System.out.println(min(f, g) + " is smaller than " + max(f, g));
        System.out.println(min(s, t) + " is smaller than " + max(s, t));
    }
}

```

Example 6: Program to demonstrate a generic method with a bounded type parameter.

Generic Classes

Recall our use of checked types when using the `Vector` class. That is, we place the class type to be stored in the object into `<>` immediately after the name of the class such as `Vector<String>`. We can create generic classes by the same means in Java 5.

Remember that prior to JDK 5 this particular method of defining classes was not available. We will now take out `Linked_List` from a previous discussion and make it generic. *Example 7* shows how this can be done. In this example it literally is as simple as placing `<T>` after the class name and then using `T` for those variables that represent the data type being stored.

```
public class GenList<T> {
    private Node<T> head;

    private class Node<T> {
        private T data;
        private Node<T> next;

        private Node(T item) {
            data = item;
            next = null;
        }
    }

    public GenList () {
        head = null;
    }

    public void dumplist() {
        Node<T> p = head;

        while ( p != null ) {
            System.out.print(p.data + " ");
            p = p .next;
        }
        System.out.println();
    }
}
```

Example 7: Class `GenList` demonstrating a generic class for a linked list.

```

public void insert(T item) {
    Node<T> n = new Node<T>(item);

    n.next = head;
    head = n;
}

public void delete(T item) {

    Node<T> cur = head, back = null;
    boolean found = false;

    while (cur != null) {
        if (cur.data.compareTo(item) == 0) {
            if (back == null)
                head = cur.next;
            else
                back.next = cur.next;

            break; // leave the loop
        } else {
            back = cur;
            cur = cur.next; // move to the next node
        }
    }
}
}

```

Example 7: Generic linked list class GenList, continued.

The programs in *Example 8*, *Example 9* and *Example 10* exercise the generic class with the Integer, String and Time classes.

```

public class TestGenListInt {

    public static void main(String[] args) {

        GenList<Integer> ll = new GenList<Integer>();

        System.out.println("\nAdding 35, 78, -45 and 0 to the list");
        ll.insert(35);
        ll.insert(78);
        ll.insert(-45);
        ll.insert(0);

        ll.dumplist();

        System.out.println("\nRemoving 0 from head of list");
        ll.delete(0); // beginning
        ll.dumplist();
        System.out.println("\nRemoving 78 from middle of list");
        ll.delete(78); // middle
        ll.dumplist();
        System.out.println("\nRemoving 35 from end of list");
        ll.delete(35); // end
        ll.dumplist();
    }
}

```

Example 8: Program to test GenList with Integer class objects.

```

public class TestGenListString {

    public static void main(String[] args) {

        GenList<String> ll = new GenList<String>();

        System.out.println("\nAdding Blue, Red, Orange and Green to the list");
        ll.insert("Blue");
        ll.insert("Red");
        ll.insert("Orange");
        ll.insert("Green");

        ll.dumplist();

        System.out.println("\nRemoving Green from head of list");
        ll.delete("Green"); // beginning
        ll.dumplist();
        System.out.println("\nRemoving Red from middle of list");
        ll.delete("Red"); // middle
        ll.dumplist();
        System.out.println("\nRemoving Blue from end of list");
        ll.delete("Blue"); // end
        ll.dumplist();
    }
}

```

Example 9: Program to test GenList with String class objects.

```

public class TestGenListTime {

    public static void main(String[] args) {

        GenList<Time> ll = new GenList<Time>();
        Time clock1 = new Time(11, 35, 0);
        Time clock2 = new Time(0, 0, 0);
        Time clock3 = new Time(16, 0, 0);
        Time clock4 = new Time(23, 59, 59);

        System.out.printf("Adding %s, %s, %s & %s to the list.",
            clock1, clock2, clock3, clock4);
        ll.insert(clock1);
        ll.insert(clock2);
        ll.insert(clock3);
        ll.insert(clock4);

        ll.dumplist();

        System.out.printf("\nRemoving %s from head of list\n", clock4);
        ll.delete(clock4); // beginning
        ll.dumplist();
        System.out.printf("\nRemoving %s from middle of list\n", clock2);
        ll.delete(clock2); // middle
        ll.dumplist();
        System.out.printf("\nRemoving %s from end of list\n", clock1);
        ll.delete(clock1); // end
        ll.dumplist();
    }
}

```

Example 10: Program to test GenList with Time class objects.