Exploring Interfaces

We previously saw how to create an interface. Java has a number of classes and methods built into the standard libraries that expect us to use interfaces in particular ways. This material covers a few of those interfaces.

Java has a static method named `sort` in the Arrays class that can be used to sort an array. Here is an example:

```
import java.util.Arrays;

public class SortDemo
{
        public static void main(String[] args)
        {
                int[] nums = new int[4];

                nums[0] = 3;
                nums[1] = 45;
                nums[2] = 1;
                nums[3] = 23;

                Arrays.sort(nums);

                for (int i : nums)
                {
                        System.out.println(i);
                }
        }
}
```

Upon compiling and running this code it will sort the array of numbers and output 1,3, 23, 45. What if we want to sort an array of something else? For example, consider the Fruit class and the SortDemo class below. Here is an attempt to make an array of fruit and sort them.

```
public class Fruit
{
        private String fruitName;
        public Fruit()
        {
                fruitName = "";
        }
        public Fruit(String name)
        {
                fruitName = name;
        }
        public void setName(String name)
        {
                fruitName = name;
        }
        public String getName()
```

```
        {
                return fruitName;
        }
}

import java.util.Arrays;
public class SortDemo
{
        public static void main(String[] args)
        {
                Fruit[] fruits = new Fruit[4];

                fruits[0] = new Fruit("Orange");
                fruits[1] = new Fruit("Apple");
                fruits[2] = new Fruit("Kiwi");
                fruits[3] = new Fruit("Durian");

                Arrays.sort(fruits);

                // Output the sorted array of fruits
                for (Fruit f : fruits)
                {
                        System.out.println(f.getName());
                }
        }
}
```

This program should compile but give an error when you run it. This error occurs because Java doesn't know how to compare two instances of the Fruit class to each other to see if one "comes after" the other when attempting to sort the array. More precisely, the Arrays.sort method has been written with the expectation that the objects passed in the array have a **compareTo method** in accordance with the **Comparable interface**. The program worked with the array of integers because Java has defined the compareTo method for Integer objects.

The Comparable interface specifies only a single method:

```
public int compareTo(Object other);
```

The compareTo method is to be written by the programmer and should return

- a negative number if the calling object "comes before" the parameter other,
- a zero if the calling object "equals" the parameter other, and
- a positive number if the calling object “comes after” the parameter other

Here is a modified version of the Fruit class that implements the Comparable interface and defines the compareTo method.

```
public class Fruit implements Comparable
{
        private String fruitName;
        public Fruit()
        {
```

```
                fruitName = "";
        }
        public Fruit(String name)
        {
                fruitName = name;
        }
        public void setName(String name)
        {
                fruitName = name;
        }
        public String getName()
        {
                return fruitName;
        }
        public int compareTo(Object obj)
        {
                // This makes sure the object is a fruit
                if ((obj != null) &&
                    (obj instanceof Fruit))
                {
                 Fruit otherFruit = (Fruit) obj; // Typecast object to a fruit
                 return (fruitName.compareTo(otherFruit.fruitName));
                          // Use String compareTo
                }
                return -1;   // Default if other object is not a Fruit
        }
}
```

This new version first checks if the object the Fruit is being compared to is another Fruit. If it is, the object is typecast to a Fruit. Then it uses the fruitName and compares it to the other fruit name using the String compareTo method. If the fruit name is alphabetically before the other fruit name then -1 is returned. If they are the same then 0 is returned. if the fruit name is alphabetically after the other fruit name then 1 is returned. This all happens in the fruitname.compareTo method.

Here is an alternate version of the Fruit class with a different compareTo implementation:

```
public class Fruit implements Comparable
{
        private String fruitName;
        public Fruit()
        {
                fruitName = "";
        }
        public Fruit(String name)
        {
                fruitName = name;
        }
        public void setName(String name)
        {
                fruitName = name;
        }
        public String getName()
        {
```

```
                return fruitName;
        }
        public int compareTo(Object o)
        {
                if ((o != null) &&
                    (o instanceof Fruit))
                {
                        Fruit otherFruit = (Fruit) o;
                        if (fruitName.length() > otherFruit.fruitName.length())
                                return 1;
                        else if (fruitName.length() <
otherFruit.fruitName.length())
                                return -1;
                        else
                                return 0;

                }
                return -1;  // Default if other object is not a Fruit
        }
}
```

This new version will sort the fruits by length of the string.

Similarly, Arrays.binarySearch will perform a binary search of the array. It requires that the array be sorted and then needs compareTo to be implemented so that it can search the array for a target.

```
System.out.println(Arrays.binarySearch(fruits, new Fruit("Orange")));
```

Returns index 3 where Orange is found.

```
System.out.println(Arrays.binarySearch(fruits, new Fruit("Banana")));
```

Returns a negative number indicating no such fruit found.

Here is one additional example. This one doesn't require implementing an interface, but show the importance of overriding certain methods inherited from the Object class.

The first is Arrays.toString(arrayname). This method outputs the array as a string. If we try it with our fruit array:

```
System.out.println(Arrays.toString(fruits));
```

We just get a bunch of weird pointers output. This is because the Arrays.toString method will invoke the toString method for each fruit object. If we just override the toString method we will get meaningful output:

```
public String toString()
{
      return fruitName;
}
```

Handling Mouse Events

As one more example, let's show how to handle a mouse event in a JFrame.
In this case, we'll just draw a circle at the coordinates where the user clicks the mouse.

Here is some starter code that just displays a window and draws a circle at 0,0:

```
import javax.swing.JFrame;
import java.awt.Graphics;
import java.awt.Color;

public class MouseCircle extends JFrame
{
        private int clickX=0, clickY=0;

        public void paint(Graphics g)
        {
                super.paint(g);
                g.setColor(Color.RED);
                g.fillOval(clickX, clickY,150,150);
        }

        public MouseCircle()
        {
                setSize(1000, 800);
                setDefaultCloseOperation(EXIT_ON_CLOSE);
                setVisible(true);       // Makes window visible
        }

        public static void main(String[] args)
        {
                MouseCircle myWindow = new MouseCircle();
        }
}
```

To get mouse events, we create an event handler or a mouse listener. This is essentially a method that will get called whenever a mouse event, like a button click, occurs. To do this we have to implement the **MouseListener** interface.

We have to import the following:

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
```

Next we need a class that implements the MouseListener interface. One option is to do this in a new class:

```
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class MyMouseListener implements MouseListener
```

The MouseListener interface requires that we implement the following methods, shown below

```
public void mousePressed(MouseEvent e)
{
}
public void mouseReleased(MouseEvent e)
{
}
public void mouseEntered(MouseEvent e)
{
}
public void mouseExited(MouseEvent e)
{
}
public void mouseClicked(MouseEvent e)
{
}
```

Even if you don't want to do anything with mouseEntered or mouseReleased events, your class must create these methods. We can output the mouse X and Y coordinates when the mouse is pressed with e.getX() and e.getY():

```
public void mousePressed(MouseEvent e)
{
      System.out.println(e.getX() + " " + e.getY());
}
```

To hook up this mouse listener with the JFrame window, go back to the MouseCircle class and add a listener with an instance of the MyMouseListener class:

```
import java.awt.event.MouseListener;
…
public MouseCircle()
{
      setSize(1000, 800);
      setDefaultCloseOperation(EXIT_ON_CLOSE);
      setVisible(true); // Makes window visible
      addMouseListener(new MyMouseListener());
}
```

Running the program and clicking the mouse will output the X/Y coordinates. Cool! But if we want to draw a circle at the given coordinates, we have a problem. The MyMouseListener class has no way to tell the MouseCircle class what coordinates we clicked on. One way to do this is to send in a reference to the class, and make a public method that redraws the screen. Here is the code:

```
import javax.swing.JFrame;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.event.MouseListener;

public class MouseCircle extends JFrame
```

```
{
	private int clickX=0, clickY=0;

	public void paint(Graphics g)
	{
		super.paint(g);
		g.setColor(Color.RED);
		g.fillOval(clickX, clickY,150,150);
	}

	public MouseCircle()
	{
		setSize(1000, 800);
		setDefaultCloseOperation(EXIT_ON_CLOSE);
		setVisible(true); // Makes window visible
		addMouseListener(new MyMouseListener(this));
	}

	public void update(int x, int y)
	{
		clickX = x;
		clickY = y;
		repaint();
	}

	public static void main(String[] args)
	{
		MouseCircle myWindow = new MouseCircle();
	}
}

import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class MyMouseListener implements MouseListener
{
	private MouseCircle window = null;

	public MyMouseListener()
	{
	}

	public MyMouseListener(MouseCircle w)
	{
		window = w;
	}

	public void mousePressed(MouseEvent e)
	{
		System.out.println(e.getX() + " " + e.getY());
		window.update(e.getX(), e.getY());
	}
	public void mouseReleased(MouseEvent e)
	{
	}
	public void mouseEntered(MouseEvent e)
	{
```

```
	}
	public void mouseExited(MouseEvent e)
	{
	}
	public void mouseClicked(MouseEvent e)
	{
	}
}
```

Since this can get a little unwieldy, a common technique is to make the JFrame window ALSO be the class that listens for mouse events. If we follow this technique then we don't need to make a separate MyMouseListener class. We just put everything into the MouseCircle class, as shown below:

```
import javax.swing.JFrame;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class MouseCircle extends JFrame implements MouseListener
{
	private int clickX=0, clickY=0;

	public void paint(Graphics g)
	{
		super.paint(g);
		g.setColor(Color.RED);
		g.fillOval(clickX, clickY,150,150);
	}

	public MouseCircle()
	{
		setSize(1000, 800);
		setDefaultCloseOperation(EXIT_ON_CLOSE);
		setVisible(true); // Makes window visible
		addMouseListener(this);
	}

	public void mousePressed(MouseEvent e)
	{
		System.out.println(e.getX() + " " + e.getY());
		clickX = e.getX();
		clickY = e.getY();
		repaint();
	}
	public void mouseReleased(MouseEvent e)
	{
	}
	public void mouseEntered(MouseEvent e)
	{
	}
	public void mouseExited(MouseEvent e)
	{
	}
```

```
	public void mouseClicked(MouseEvent e)
	{
	}

	public static void main(String[] args)
	{
		MouseCircle myWindow = new MouseCircle();
	}
}
```

Finally, if you prefer the specific class approach, you can make a MouseListener as an inner class inside MouseCircle. A common approach is to use what is called an anonymous inner class, in which you declare a class (but have no name for it) right in the place where the class would normally be passed by reference. Since the class is declared inside the MouseCircle class, it has access to all the private MouseCircle variables.

```
import javax.swing.JFrame;
import java.awt.Graphics;
import java.awt.Color;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

public class MouseCircle extends JFrame
{
	private int clickX=0, clickY=0;

	public void paint(Graphics g)
	{
		super.paint(g);
		g.setColor(Color.RED);
		g.fillOval(clickX, clickY,150,150);
	}

	public MouseCircle()
	{
		setSize(1000, 800);
		setDefaultCloseOperation(EXIT_ON_CLOSE);
		setVisible(true); // Makes window visible
		addMouseListener(new MouseListener()
		 {
			public void mousePressed(MouseEvent e)
			{
				System.out.println(e.getX() + " " + e.getY());
				clickX = e.getX();
				clickY = e.getY();
				repaint();
			}
			public void mouseReleased(MouseEvent e)
			{
			}
			public void mouseEntered(MouseEvent e)
			{
			}
			public void mouseExited(MouseEvent e)
			{
```

```
                    }
                    public void mouseClicked(MouseEvent e)
                    {
                    }
                }
            );
        }

        public static void main(String[] args)
        {
                MouseCircle myWindow = new MouseCircle();
        }
}
```

This technique looks a little odd, but is common with GUI applications where we want to make a short, quick and dirty event handler.

Java 8 incorporates a feature called lambda expressions. You will study this idea more in CSCE A331, programming languages. Basically, it is a nameless function. You can insert the function in place of a class that should implement a specific, single method. We can't use it in this case since we are required to implement multiple methods, but you will see this in action to implement button clicks or other event handlers.