Computer Science 252
Problem Solving with Java
The College of Saint Rose
Fall 2013

Topic Notes: Interfaces

Our next topic is a Java feature called *interfaces*. This topic is closely related to our recent discussion of custom classes, which gave us good examples of *object-oriented design*.

This topic is also only tangentially related to an upcoming topic: Java's Graphical User Interface mechanisms. That is, while this topic "Interfaces" and the upcoming topic "User Interfaces" share part of their name, the connection between the two topics is not very strong.

For now, we will study the strong connection between interfaces and object-oriented design.

In our discussion of design, we saw that an important goal was to hide the details of how the behavior of a class was implemented by keeping its instance variables private and then defining public methods that provided the ability to change or determine a limited set of aspects of the object's state. For example, when you modified the T-Shirt example to have more graphical components, you were able to do that entirely within the `TShirt` class, without any changes to the `Drag2Shirts` class. As long as you maintained a consistent *public interface* to the `TShirt` class, that is, the method signatures of the `public` methods were unchanged, the `Drag2Shirts` class would still be able to use it in the same way, even though the `TShirt` class was now doing something different than it was before.

The public interface of the objects of a Java class can be concisely described by listing all of its methods and describing their parameters. That is why we ask you to include such informations in your designs.

Your designs are (quite deliberately) not actual Java code. It turns out, however, to be quite useful to write a description of a classes interface in a form that is interpretable as Java code by the computer. Accordingly, Java's rules of syntax permit such a specification.

To illustrate the use of this new Java mechanism we will look at a fancier version of the laundry sorter game you wrote. This version does a better job with object orientation and allows multiple kinds of laundry items to be sorted.

See Example: ShirtsAndPants

In this case, `Laundry.java` is an *interface specification* rather than a regular Java class.

- Like a class definition, an interface specification is constructed out of a header followed by a body enclosed within curly braces.
- The body consists of a list of method headers. Each header is terminated by a semicolon (rather than being followed by a method body).

With this interface included in our program, we can now construct a more advanced version of the laundry sorter.

Much of this program is very similar to what you did for the lab. However, instead of dragging around a `FilledRect` and a `FramedRect`, we are dragging around either a `TShirt` or `Pants` object.

The code that handles the dragging, checking for the correct basket, *etc.*., is otherwise unchanged from my reference solution.

Herein lies the power of the interface construct. As long as we only call methods that are part of the interface, we can interchange any objects that implement that interface and the rest of our program can remain unchanged. Just like we could call `move`, `moveTo`, and `contains` on the `FilledRect` and `FramedRect` that represented our laundry swatches, we can substitute in the objects from this example.

Interfaces let us write such highly flexible code.

In our new laundry example, two classes of laundry defined. `TShirt`s and `Pants`. Since they both contains all the methods listed in our `Laundry` interface, we can explicitly tell Java this in the headers of the class declarations

```
   public class TShirt implements Laundry
```

So long as we restrict ourselves to the methods defined by the `Laundry` interface, it doesn't matter what type of laundry we come up with in the future (maybe we want to add `Shorts` next spring), we can continue to use any program that uses the `Laundry` interface without changing it!

To tell Java that we want to use objects that implement the `Laundry` interface in our laundry sorter, we can replace the declaration of the item variable in the window controller with a declaration of the form:

```
private Laundry item;
```

When we write a declaration, remember that we are telling Java what sort of object the name may be associated with. The declaration “`private Laundry item`” tells Java that item may be associated with an object of any class that implements the `Laundry` interface.

So, Java will allows us to say either:

```
item = new TShirt( ... );
```

or

```
item = new Pants( ... );
```

since we have taught Java how to construct `TShirt` and `Pants` objects. It allows us to assign the returned reference to either of those objects to the same variable `item` of type `Laundry` since in addition to being `TShirt` or `Pants` objects, they are also `Laundry` objects (we told Java they were when we said "`implments Laundry`".

On the other hand, java will not allow us to say:

```
item = new Laundry( ... );
```

If you think about this a bit, hopefully it makes a lot of sense. In the specification of the interface `Laundry`, we did not include a constructor. That is because in order to construct an object, we need to know the details of how its state will be represented. The whole point of an interface, however, is to focus on the interface through which the state can be manipulated while ignoring the details of how the state is represented.

On the other hand, given that Java knows that "item" will only be associated with things that implement the `Laundry` interface, it will let us write commands like:

```
      item.moveTo(...);
```

Further note that Java will not allow us to assign any old object with the right methods to the "item" variable. In particular, even if the methods in the `Laundry` interface were a subset of those provided by `FilledRect`s, we could not say:

```
      item = new FilledRect( ... );
```

since the declaration of the class `FilledRect` does not say:

```
   public class FilledRect implements Laundry {

   }
```

Also, note that we can use interface names as type names in parameter declarations.

Handling Keyboard Events

Java interfaces will arise in a number of contexts as we extend our graphical programs beyond the simple "canvas" graphics that can respond only to mouse events. Our first such example will allow us to have our programs respond to keyboard events. In particular, we will first consider the arrow keys.

See Example: HandlingArrowKeys

Parts of this program are very familiar – we have a class that `extends WindowController`, it draws some Objectdraw `Text` objects in the `begin` method. We focus on the new items:

First, at the top, we have two new `import` statements:

```
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
```

These are the items from the Java API that we will need to use. We could have done the more generic

```
import java.awt.event.*;
```

which would `import` everything from `java.awt.event`, including the two we need, but it is generally good practice to `import` only exactly what you need to avoid potential unexpected name conflicts.

Next, we tell Java that our class `implements KeyListener`. `KeyListener` is an interface in the Java API, which requires that we define three methods:

```
   public void keyTyped(KeyEvent e);
   public void keyPressed(KeyEvent e);
   public void keyReleased(KeyEvent e);
```

We can see the details at

Java API Documentation: java.awt.event.KeyListener at
`http://docs.oracle.com/javase/7/docs/api/java/awt/event/KeyListener.html`

Basically, we are saying that any class that implements this interface, thereby promising to provide those three methods, can be used to handle keyboard events. That is, we can tell Java to use these methods in this class any time a key is pressed, released, or typed (a press followed by a release).

Note that unlink the "onMouse" methods, where we could specify only those we cared about for a particular program, we have to provide all three of these to be able to satisfy the `KeyListener` interface.

The methods themselves look a lot like the mouse events we've been working with all along. Instead of receiving a `Location` as their parameter, however, they receive a `KeyEvent`. This `KeyEvent` object can then be used to determine the details of the key event that has occurred, and our program can react accordingly.

We are just using a few of the capabilities of the `KeyEvent` class. We call its `getKeyCode` method to retrieve an `int` code that tells is which key was pressed/typed/released. The `KeyEvent` class also provides a large collection of named constants called *virtual key codes*. In this example, we are interested only in the arrow keys, whose codes are `KeyEvent.VK_UP`, `KeyEvent.VK_DOWN`, `KeyEvent.VK_LEFT`, and `KeyEvent.VK_RIGHT`.

We can see all of the methods and virtual key codes at:

Java API Documentation: java.awt.event.KeyEvent at
`http://docs.oracle.com/javase/7/docs/api/java/awt/event/KeyEvent.html`

Lastly, we need to "attach" this `KeyListener` class to the windows and/or other objects whose key events we wish to receive. In this case, we add it to the `WindowController` class itself (actually, the `Applet` that underlies it) and to our `DrawingCanvas` we know as `canvas`, using the `addKeyListener` method of those objects.