CMSC 131: Chapter 13 (Supplement)
Interfaces II

Motivation (recap)

Recall that interfaces are intended to address two opposing goals:
(1) Strong Typing, and (2) General-Purpose Functions

The Problem: Strong typing implies it is impossible to write a generic sorting function.

Recall: We want to be able to write many methods with the same basic functionality, e.g.,
sortInts(), sortDoubles(), sortStrings(), sortDates(), sortRationals(), ...

Java Interface: allows programmers to write general purpose functions.

To make this work: Java provides mechanism (an interface) for general-purpose functions
(like sort) to specify what behavior they require from specific classes (like X).

Java Interfaces Summary (Review)

Defining a Java Interface:

```java
public interface Y {
    public void someMethod(int z);
    public int anotherMethod();
}
```

An interface is not a class. Classes must promise to implement certain methods defined in
the interface.

Implementing an Interface:

```java
public class X implements Y {
    // ...(instance data and other methods)...
    public void someMethod(int z) { /* give implementation here */ }
    public int anotherMethod() { /* give implementation here */ }
}
```

Now, we may use an X any place that an object of type Y is expected.
Example: SelectorInt Class

Let us begin with an example in which interfaces would be helpful.

SelectorInt: Your boss asks you to write a class SelectorInt. This class has three static methods:

\[ \min(x_1, x_2, x_3): \] \( x_1, x_2, x_3 \) are integers. This returns an integer 1, 2, or 3 depending on which is smallest: \( x_1, x_2, \) or \( x_3. \)

\[
\begin{align*}
\text{SelectorInt.min(1 23, 45, 79):} & \quad \text{returns 2 (since 45 is smallest)} \\
\text{SelectorInt.min(11, -4, -18):} & \quad \text{returns 3 (since -18 is smallest)} \\
\text{SelectorInt.min(13, 13, 25):} & \quad \text{returns either 1 or 2 (we don't care)}
\end{align*}
\]

\[ \max(x_1, x_2, x_3): \] returns an integer 1, 2, or 3 depending on which is largest, \( x_1, x_2, \) or \( x_3. \)

\[ \text{median}(x_1, x_2, x_3): \] returns an integer 1, 2, or 3 depending on which is in the middle of the order, \( x_1, x_2, \) or \( x_3. \)

For the rest of the lecture, we'll just consider \( \min, \) since the others are similar.

SelectorInt Implementation

```java
public class SelectorInt {
    /* Returns the position of the minimum element: 1, 2, or 3 */
    public static int min(int x1, int x2, int x3) {
        if (x1 < x2) { // x2 is not min, it's either x1 or x3
            if (x1 < x3) return 1;
            else return 3;
        } else { // x1 is not min, it's either x2 or x3
            if (x2 < x3) return 2;
            else return 3;
        }
    }
    // other methods (min, median) omitted...
}

public class SelectorDemo {
    public static void main(String[] args) {
        int result = SelectorInt.min(23, 12, 74);
        System.out.println("Position of Min: " + result);
    }
}
```
String Selector and Beyond

Success: Your class SelectorInt is a big hit.

Bad News: Your boss now wants you to write Selector objects for many other types: Strings, Dates, Rationals, phone numbers, names, ...

SelectorString: Should have virtually the same structure, but we cannot use "x1 < x2" on strings. We need to use "x1.compareTo(x2)". In fact, all these selectors would be almost the same. All that changes is how objects are compared to each other.

Question: Is there some way to write only one Selector class, and have it work for all these objects? What we need is a generic Selector class.

Designing a Generic Selector

Uniform Behavior: Because different classes have different ways of doing comparisons, we must have them all agree to do comparisons in one unified way.

isLessThan: Consider two objects, x1 and x2, of some class. To implement the tests, x1 < x2, we require that this class implements the following method:

\[ x1.\text{isLessThan}(x2) \] Returns true if x1 < x2 and false otherwise

If we succeed, we can design a selector for any class that promises to provide this method.

Generic Selector

Recap of where we are:
- We want to design a single generic Selector class that works for many different types of objects.
- Selector needs each object to provide a unified way to compare instances of the given class.
- To do this, we require that any object for which we can build a Selector must provide us with a comparison method:
  \[ x1.\text{isLessThan}(x2) \]
  where x1 and x2 are instances of this object.
- Any class that provides these two comparison methods is said to be Testable.
- Thus, rather than working with just int, or String, or Date, the method Selector.min can work with any Testable object.
(Old) SelectorInt

```java
public class SelectorInt {
    /* Returns the position of the minimum element: 1, 2, or 3 */
    public static int min( int x1, int x2, int x3 ){
        if ( x1 < x2 ){ // x2 is not min, it's either x1 or x3
            if ( x1 < x3 ) return 1;
            else return 3;
        } else { // x1 is not min, it's either x2 or x3
            if ( x2 < x3 ) return 2;
            else return 3;
        }
    }
    // ...other methods (min, median) omitted...
}
```

(New) Generic Selector

```java
public class Selector {  // Returns the position of the minimum element: 1, 2, or 3 */
    public static int min( Testable x1, Testable x2, Testable x3 ){  
        if ( x1.isLessThan( x2 ) ) { // x2 is not min, it's either x1 or x3
            if ( x1.isLessThan( x3 ) ) return 1;
            else return 3;
        } else { // x1 is not min, it's either x2 or x3
            if ( x2.isLessThan( x3 ) ) return 2;
            else return 3;
        }
    }
    // ...other methods (min, median) omitted...
}
```

Testable: A Java Interface

```java
public interface Testable {  // Returns True if this object is less than x */
    public Boolean isLessThan( Object x );
}
```

Making a Testable Integer

**A Testable Integer:** Since an `int` is a primitive type, we create a `wrapper` object.

**MyInteger:** Stores a single `int` as data. It "implements Testable" by providing the implementation of `isLessThan()`.

```java
/* A Testable int wrapper */
public class MyInteger implements Testable {  
    int data;

    public MyInteger( int d ) { data = d; }

    public String toString() { return String.valueOf( data ); }

    public boolean isLessThan( Object x ){  
        MyInteger m = ( MyInteger ) x; // cast x to MyInteger
        return ( data < m.data );
    }
}
```
Dissecting `MyInteger.isLessThan()`

Implementing `MyInteger.isLessThan()`: Why did we need to cast `x` to `MyInteger`?

```java
public boolean isLessThan(Object x) {
    MyInteger m = (MyInteger) x;    // cast x to MyInteger
    return (data < m.data);
}
```

Alternatives that do not work:

- Avoid the cast?

```java
public boolean isLessThan(Object x) { return (data < x.data); }
```

Since `x` is not `MyInteger` (it is `Object`) we cannot access `x.data`.

- Declare `x` to be `MyInteger`?

```java
public boolean isLessThan(MyInteger x) { return (data < x.data); }
```

This does not match the `isLessThan()` signature of the interface, and so Java will issue a compile error that you have not implemented the interface properly.

Using `MyInteger` in `Selector`

Using `Selector` on `MyInteger`:

Because `MyInteger` implements the `Testable` interface, we can call:

```java
Selector.min(Testable x1, Testable x2, Testable x3);
```

where `x1, x2, x3` are of type `MyInteger`.

```java
public static void main(String[] args) {
    MyInteger x1 = new MyInteger(23); // create three MyIntegers
    MyInteger x2 = new MyInteger(12);
    MyInteger x3 = new MyInteger(74);

    System.out.println("x1 = " + x1 + "\n" +
    "x2 = " + x2 + "\n" +
    "x3 = " + x3);
    int result = Selector.min(x1, x2, x3);
    System.out.println("Position of Min: \" + result);
}
```
Making a Testable String

Next, let's see how we can apply `Selector.min()` to Strings. We need to make a Testable String.

`MyString`: We create a String wrapper, and define `isLessThan()`. Recall that Strings are compared using `compareTo()`.

```java
/* A Testable String wrapper */
public class MyString implements Testable {
    String str;

    public MyString( String s ) { str = new String( s ); }
    public String toString() { return str; }
    public boolean isLessThan( Object x ) {
        MyString s = ( MyString ) x; // cast x to MyString
        return str.compareTo( s.str ) < 0;
    }
}
```

Using `MyString` in `Selector`

Using `Selector.on` `MyString`:

Because `MyString` implements the Testable interface, we can call:

```java
Selector.min( Testable x1, Testable x2, Testable x3 );
```

where `x1, x2, x3` are of type `MyString`.

```java
public static void main( String[] args ) {
    MyString s1 = new MyString( "Bob" );
    MyString s2 = new MyString( "Carol" );
    MyString s3 = new MyString( "Alice" );

    System.out.println( "s1 = " + s1 + "\n" +
                        "s2 = " + s2 + "\n" +
                        "s3 = " + s3 );
    int result = Selector.min( s1, s2, s3 );
    System.out.println( "Position of Min: " + result );
}
```