CS 213 – Software Methodology

Sesh Venugopal

Interfaces - Fundamentals

Comparing for inequality in a library module

```
public class Searcher {
   ...
   public static<T> boolean
   binarySearch(T[] list, T target) {
       ...
       list[index].___?___target

       ...
   }
   ...
}
```

How to compare for inequality? All we know Is `T` is some `Object`, but `Object` does not define an inequality comparison method

Want to somehow specify that Ts are not *any* objects, but only those objects that have some known inequality comparison method

AND, this specification MUST be checkable by compiler, so that (a) our `binarySearch` will compile, and (b) the client code's call to this method will be guaranteed to send in required type of object

How to specify a T type with inequality support?

```
public class Searcher {
   ...
   public static<T> boolean
   binarySearch(T[] list, T target) {
       ...
       list[index].___?___target

       ...
   }
   ...
}
```

A class is a user-defined type, e.g. `Point` and `ColoredPoint` are types introduced by the program, which can be checked by the compiler (and appropriately matched at run time)

But we (library designer) can't implement a new class type instead of T that has, say, a `compareTo` method because that would take away the generic nature of the type, and prevent clients from sending different kinds of objects at different times to this `binarySearch` method (each kind of object implements its own custom version of comparing for inequality)

How to specify a T type with inequality support?

```
public class Searcher {
   ...
   public static<T> boolean
   binarySearch(T[] list, T target) {
       ...
       list[index].___?___target

       ...
   }
   ...
}
```

Solution is to make like we are defining a new class (type), with an inequality method, but stop short of actually implementing the method body – this is an INTERFACE

e.g. `java.lang.Comparable` interface, which defines a `compareTo` method, without a body:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

No curly braces!!

Then it's up to the client to fit a matching class with the `compareTo` method body custom filled in as needed

What interface to use with `binarySearch` method?

```
public class Searcher {
   ...
   public static<T> boolean
   binarySearch(T[] list, T target) {
       ...
       list[index].___?___target

       ...
   }
   ...
```

We have the option of using any of the interfaces defined in Java, or roll our own if none of those fit our need

In our `Searcher` example, `Comparable` would be a perfect fit

```
public class Searcher {
   ...
   public static <Comparable<T>> boolean
   binarySearch(Comparable<T>[] list, Comparable<T> target) {
       ...
       list[index].compareTo(target)
       ...
   }
   ...
```

WILL NOT COMPILE
(not proper generic type syntax)

How to specify that `binarySearch` expects `Comparable<T>` type objects?

```
public class Searcher {
   ...
   public static <T extends Comparable<T>> boolean

   binarySearch(T[] list, T target) {
       ...
       list[index].compareTo(target)
       ...
   }
   ...
```

Type `T` is not just any class, but one that <u>implements</u> the `java.lang.Comparable<T>` interface, or extends a class (any number of levels down the inheritance chain) that implements the `java.lang.Comparable<T>` interface

Objects that can match `binarySearch` requirement of `T extends Comparable<T>`

```
public class Point implements Comparable<Point> {
   ...
   public int compareTo(Point other) {

       int c = x - other.x;
       if (c == 0) {
          c = y - other.y;
       }
       return c;

   }
   ...
```

Type `Point` is not just any class, but one that <u>implements</u> the `java.lang.Comparable<Point>` interface

Objects that can match `binarySearch` requirement of `T extends Comparable<T>`

```
public class ColoredPoint extends Point {
   ...
   public int compareTo(Point other) { // Inherited

       int c = x - other.x;
       if (c == 0) {
          c = y - other.y;
       }
       return c;

   }
   ...
```

Type `ColoredPoint` is not just any class, but one that extends a class (`Point`) that implements `java.lang.Comparable<Point>`

By virtue of extending `Point`, `ColoredPoint` implicitly implements the `Comparable<Point>` interface, equivalent to:

```
public class ColoredPoint extends Point implements Comparable<Point>
```

Implicit interface – Public members of a class

The term "interface" GENERALLY refers to the means by which an object can be manipulated by its clients – in this sense the public fields and methods of an object comprise its implicit interface.

For example, public methods `push`, `pop`, `isEmpty` (as well as constructors) in a `Stack` implicitly define its interface – these methods/constructors will be used by clients to create and manipulate stacks

Explicit Interface

Java provides a way (keyword `interface`) to define an explicit interface that can be implemented (keyword `implements`) by classes

```
public interface I { . . . }
public class X implements I { . . . }
```

The (generic) `Comparable` interface is defined in `java.lang` package

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Prescribes a single, `compareTo` method, but there is no method body, just a semicolon terminator

For method `compareTo`, keywords `public` and `abstract` are omitted by convention (redundant if written)

Interfaces - Properties

Properties of interfaces:

1. An interface defines a new type that is tracked by the compiler
2. All fields in an interface are constants: implicitly public, static, and final
3. Prior to Java 8, all interface methods were implicitly public and abstract (no method body)
4. As of Java 8, interfaces can also include default and static methods (fully implemented) – these need to be public
5. As of Java 9, interfaces can also have fully implemented private methods (static or non static)
6. When a class implements an interface, it must implement every single abstract method of the interface
7. An interface J can extend another interface I, in which case I is the super interface and J is its sub interface

Interfaces - Properties

Properties of interfaces - continued:

8. A class man implement multiple interfaces

```
public class X implements I1, I2, I3 { ... }
```

9. A subclass implicitly implements all interfaces that are implemented by its superclass

```
public class Point implements Comparable<Point> { ... }
public class ColoredPoint extends Point
          implements Comparable<Point> { ... }
```

implicit (writing it out is ok too)

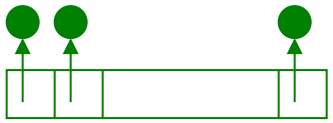
10. An interface may be generic, but this does not mean an implementing class must use its own type to match the generic type – see the ColoredPoint example above

Using `java.lang.Comparable`

```
public class Point
  implements Comparable<Point> {
    . . .

    public int compareTo(Point other) {
        int c = x - other.x;
        if (c == 0) {
            c = y - other.y;
        }
        return c;
    }

}
```

```
public class Widget
  implements Comparable<Widget> {
    . . .

    public int compareTo(Widget other) {
        float f = mass - other.mass;
        if (f == 0) return 0;
        return f < 0 ? -1 : 1;
    }

}
```

Array of `Point` *objects*



target `Point`

```
public static <T extends Comparable<T>>
    boolean binarySearch(T[ ] list,
                         T target) {
        ...
        int c = target.compareTo(list[i]);
        ...
    }
}
```

Array of `Widget` *objects*

target `Widget`

Interface javafx.event.EventHandler

```
public interface EventHandler<T extends Event> {
   void handle(T event);
}
```

javafx.scene.control.ButtonBase defines this method:

```
public void setOnAction(EventHandler<ActionEvent> value) {
   ...
}
```

The parameter to this method is any object that implements the EventHandler<ActionEvent> interface.

javafx.scene.control.Button is a subclass of ButtonBase:

```
f2c.setOnAction(new EventHandler<ActionEvent>() {
    public void handle(ActionEvent e) {...}
});
```

Anonymous class that implements the EventHandler<ActionEvent> interface

Object created by calling the default constructor of the anonymous class