CS 213 – Software Methodology

Spring 2023

Sesh Venugopal

Feb 13

Interfaces - 2

What interface to use for T in 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 fits 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)
       ...
   }
   ...
```

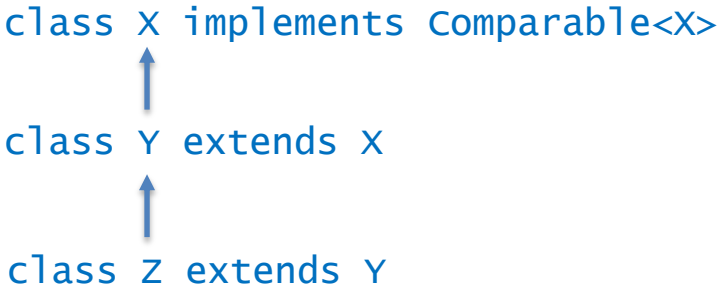
WILL NOT COMPILE
(not proper generic type syntax)

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

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` stands for a class that implements the `java.lang.Comparable<T>` interface, OR extends a class (to any number of levels down the inheritance chain) that implements the `java.lang.Comparable<T>` interface



X, Y, and Z will all match T

Example object types that match the `binarySearch` requirement of `T extends Comparable<T>`

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 implements the `java.lang.Comparable<Point>` interface

Since `Point implements Comparable<Point>` it MUST implement the `compareTo` method specified by the `Comparable` interface, with `Point` as the parameter. Otherwise, there will be compile-time error.

On the other hand, if Point implements the compareTo method specified by the Comparable interface but omits implements Comparable<Point> then it would NOT match `T extends Comparable<Point>` and it can't be sent as an argument to binarySearch

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

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

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);
}
```

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

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

Interface 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

Properties of interfaces - continued:

8. A class may 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 require an implementing class to match the generic type with itself – see the ColoredPoint example above

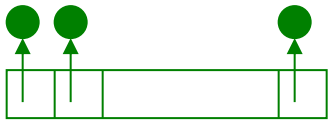
Two Use Cases of Java SDK Interfaces

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);
}
```

javax.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.

javax.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

When to Use Interfaces

Use #1:
To Make Classes Conform to a Specific Role Used in External Context

Classes – Conform to Specific External Role

Often,
a specialized role needs to be specified
for some classes in an application (e.g. comparing for ==, >, <),
and given a type name (.e.g. `Comparable, EventHandler`)

The type name is the interface name,
and the role is the set of interface methods.

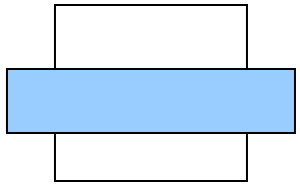
You can think of an interface as
a filter that is overlaid on a class.

Depending on the context,
the class can be fully itself (class type)
or can adopt a subset, specialized role (interface type)

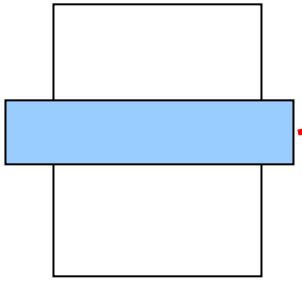
Specialized Role For Classes

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

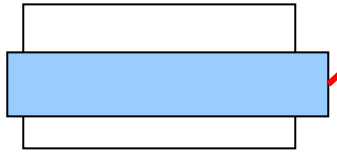
class X implements Comparable<X>



class Y implements Comparable<Y>



class Z implements Comparable<Z>



methodM will admit any object, so long as it is Comparable, and it knows the admitted object ONLY as Comparable – that is, the filter is blind to all other aspects of the object type (X, or Y, or Z) but the Comparable part

class U

```
static
<T extends Comparable<T>>
void methodM(T c) {
   ...
}
```

The implementor of methodM in class U may call the compareTo method on the parameter object c, without knowing anything about the argument except that it will be guaranteed to implement compareTo

Use #2:
To define a single type that gathers functionality common to classes that are not in an inheritance hierarchy

Type for Classes with Common Behavior

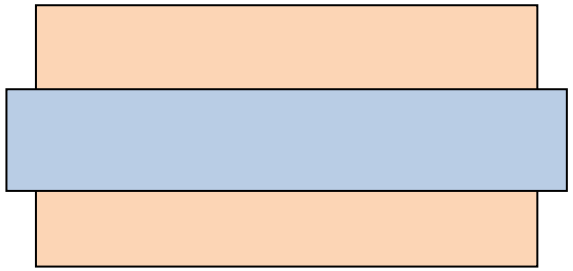
Zebras, Horses and Donkeys can all trot, gallop, and snap (common behavior)

In a simulation with many instances of each, you may want to evoke one or more of these behaviors in a randomly selected instance or group, without regard to what exact specimen is targeted – grouping these behaviors under a new type meets this need

```
public interface Equine {
   void trot();
   void gallop();
   void snap();
 }
```

```
class Zebra implements
      Equine
```

```
class Horse implements
      Equine
```



```
class Donkey implements
      Equine
```

Polymorphism using interface type

A collection (e.g. `ArrayList`) might have a combination of zebras, donkeys and horses

```
ArrayList<Equine> equines = new ArrayList<>();
equines.add(new Zebra());
equines.add(new Horse());
...
```

Now you can apply any of the common behaviors to instances of the collection, without regard to the actual type of animal (*no need to check what actual type it is*):

```
for (Equine eq: equines) {

  eq.trot();
   ...
}
```


actual behavior is executed on runtime instance

This is polymorphism via an interface type – common behavior executed on objects with same interface (static) type, but the way the behavior is executed is automatically determined by binding to the run time type ("shape" of object changes automatically, hence poly "morph" ism.)

Use #3:
To Set Up an Invariant Front for Different Implementations of a Class (Plug and Play)

As a Front for Different Implementations (Plug and Play)

Stack structure

```
package util;

public class Stack<T> {
   private ArrayList<T> items;
   public Stack() {...}
   public void push(T t) {...}
   ...
}
```

Stack client

```
package apps;
import util.*;
public class SomeApp {
   ...
   Stack<String> stk =
       new Stack< >();
   stk.push("stuff");
   ...
}
```

Plug and Play

The `util` group wants to provide an alternative stack implementation that uses a linked list instead of an `ArrayList`.

In the process, it changes the name of the push method:

The client needs to make appropriate changes in the code in order to use the LL alternative:

```
package util;

public class LLStack<T> {
    private Node<T> items;
    public LLStack() {...}
    public void llpush(T t) {...}
    ...
}
```

```
package apps;
import util.*;
public class SomeApp {
    ...
    LLStack<String> stk =
        new LLStack< >();
    stk.llpush("stuff");
    ...
}
```

To switch between alternatives, client has to make several changes.
Functionality (WHAT can be done - push) bleeds into implementation (HOW it can be done – ArrayList/Linked List) in the push/llpush methods.

Stack Alternatives: Better solution

Define a Stack Interface

Stack interface

```
package util;

public interface Stack<T> {
   void push(T t);
   T pop();
   ...
}
```

ArrayList version

```
package util;

public class ALStack<T>
implements Stack<T> {
   private ArrayList<T> items;
   public ALStack() {...}
   public void push(T t) {...}
   public T pop() {...}
   ...
}
```

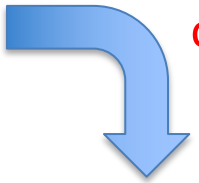
Linked List version

```
package util;

public class LLStack<T>
implements Stack<T> {
   private Node<T> items;
   public LLStack() {...}
   public void push(T t) {...}
   public T pop() {...}
   ...
}
```

Client Use of Stack Interface

```
package apps;

public class SomeApp {
    ...
    Stack<String> stk =
        new ALStack<String>();
    stk.push("stuff");
    ...
}
```

Use interface `Stack` for static type



To use other stack, only one change – in **new**

```
package apps;

public class SomeApp {
    ...
    Stack<String> stk =
        new LLStack<String>();
    stk.push("stuff");
    ...
}
```

Plug and Play – Example 2

In an application that does stuff with lists, there is a choice of what kind of list to use:

`ArrayList` used, statically typed to `ArrayList`:

```
ArrayList list = new ArrayList( );
. . .
list.<ArrayList method>(…)
. . .
```

OR

`ArrayList` used, statically typed to `java.util.List` (interface)

```
List list = new ArrayList( );
. . .
list.<List method>(. . .)
. . .
```

Plug and Play Example 2

Consider later switching to a different implementation of a list, say `java.util.LinkedList`.
The `LinkedList` class also implements the `List` interface.

In the version where `list` is statically typed to `ArrayList`:

```
LinkedList                LinkedList
ArrayList list = new ArrayList( );
. . .
list.<ArrayList method>(…)
. . .                 ?
```

What if this method is not in the `LinkedList` class?

Need to check *all* places where a `list`.<method>(…) is called. Then keep it as it is (same functionality is in `LinkedList`), or change it to an equivalent `LinkedList` method (if one exists), and if not, somehow devise equivalent code.

Plug and Play Example 2

Consider later switching to a different implementation of a list, say `LinkedList`. The `LinkedList` class also implements the `List` interface.

In the version where `list` is statically typed to `ArrayList`:

```
                    LinkedList
List list = new ArrayList( );
. . .
list.<List method>(…)
. . .
```

Just replace `new ArrayList()` with `new LinkedList()`
No other changes needed

Plug and Play:
Using a static interface type (for reference variable)
to switch implementations (for object at run time)
is a kind of interface polymorphism

Use #4:
As a workaround for multiple inheritance

Workaround for Multiple Inheritance

```
public class Phone {
   public void makeCall(...) {...}
   public void addContact(...) {...}
   ...
}

public class MusicPlayer {
   public Tune getTune(...) {...}
   public void playTune(...) {...}
   ...
}
```

Want a class to implement a device that is both a phone and a music player:

```
public class SmartPhone
extends Phone, MusicPlayer {
   public void makeCall(...) {...}
   public void addContact(...) {...}
   public Tune getTune(...) {...}
   public void playTune(...) {...}
   ...
}
```

Can't extend more than one class!

Workaround for Multiple Inheritance

```
public class Phone {
   public void
      makeCall(...) {...}
   public void
      addContact(...) {...}
   ...
}
```

```
public class MusicPlayer {
   public Tune
      getTune(...) {...}
   public void
      playTune(...) {...}
   ...
}
```

Workaround is to define at least one of the types as an interface:

```
public interface MusicPlayer {
   Tune getTune(...);
   void playTune(...);
   ...
}
```

Drawback is `getTune` and `playTune` will have to be re-implemented in `SmartPhone` instead of being reused from `MusicPlayer`

```
public class SmartPhone
  extends Phone
  implements MusicPlayer {
   public void makeCall(...) {...}
   public void addContact(...) {...}
   public Tune getTune(...) {...}
   public void playTune(...) {...}
   ...
}
```