

ECS510

Algorithms and Data Structures in an
Object Oriented Framework
“ADSOOF”

Interface Types and Generics

The `PricedObject` examples
and the “Why a Duck?” examples

Interface Types

- A Java interface type is given by using the keyword `interface` instead of `class`
- It only contains method headers
- A class `implements` an interface
- While a class can only extend one other class, it can implement any number of interfaces
- A class which implements an interface must provide code for all its methods
- Interfaces are used to write generalised code

PricedObject

- The interface

```
interface PricedObject
{
    public int getPrice();
}
```

provides us with a type which can refer to any object which has a `getPrice()` method (its class must be declared as `implements PricedObject`)

- This means we can write generalised code, the same code can be used for a variety of objects so long as they have in common the method `getPrice()`

Generalised code

- `class DrinksMachine implements PricedObject`
- `class ScrabbleWord implements PricedObject`
- Then we can have:

```
public static PricedObject  
    cheaper(PricedObject m1,PricedObject m2)  
{  
    if(m1.getPrice()  
        <m2.getPrice()  
        )  
        return m1;  
    else  
        return m2;  
}
```

- Works on `DrinksMachine` and `ScrabbleWord` objects

A Problem

- A `PricedObject` variable (including a method parameter) can refer to a `DrinksMachine` object or a `ScrabbleWord` object
- So can an `ArrayList<PricedObject>` variable refer to an `ArrayList<DrinksMachine>` object or an `ArrayList<ScrabbleWord>` object?
- No
- Why not?

Baskets of Fruit

- Consider
 - `Apple` implements `Fruit`
 - `Banana` implements `Fruit`
- If we have a `Basket<Fruit>` object we can add `Apple` or `Banana` objects to it
- If we have a `Basket<Banana>` object we can only add `Banana` objects to it
- So a `Basket<Fruit>` variable should not refer to a `Basket<Banana>` object, that prevents us from writing code which wrongly adds `Apple` objects to it

Generalised Collection Methods

- What about

```
static PricedObject mostExpensive(ArrayList<PricedObject> a)
{
    PricedObject highestSoFar = a.get(0);
    for(int i=1; i<a.size(); i++)
    {
        PricedObject next = a.get(i);
        if(highestSoFar.getPrice()<next.getPrice())
            highestSoFar = next;
    }
    return highestSoFar;
}
```

- Isn't this just the sort of generalised code we would like to write?
We want to be able to pass `ArrayList<DrinksMachine>` and `ArrayList<ScrabbleWord>` arguments to it

Bounded Generic Types

- This is Java's solution:

```
public static <T extends PricedObject> T mostExpensive(ArrayList<T> a)
{
    T highestSoFar = a.get(0);
    for(int i=1; i<a.size(); i++)
    {
        T next = a.get(i);
        if(highestSoFar.getPrice()<next.getPrice())
            highestSoFar = next;
    }
    return highestSoFar;
}
```

- A type variable declared as `extends` a type can be set to that type or any subtype of it, and methods from that type can be called on variables of the type of the type variable

Anonymous Bounded Variables

- Here is an alternative solution in Java:

```
public static PricedObject
    mostExpensive(ArrayList<? extends PricedObject> a)
{
    PricedObject highestSoFar = a.get(0);
    for(int i=1; i<a.size(); i++)
    {
        PricedObject next = a.get(i);
        if(highestSoFar.getPrice()<next.getPrice())
            highestSoFar = next;
    }
    return highestSoFar;
}
```

- This can be done when we don't need a named typed variable. Instead of declaring a type variable as previously, it says that the element type of the ArrayList is either PricedObject or a subtype of PricedObject, and refers to objects of its type through local variable of typed PricedObject

Inferring Type Variable Values

- Using the idea of bounded type variables it is better to declare the method `cheaper` as:

```
public static <T extends PricedObject> T cheaper(T m1,T m2)
{
    if(m1.getPrice()<m2.getPrice())
        return m1;
    else
        return m2;
}
```

- Then when we have:

```
DrinksMachine mach3 = PricedObjectOps.cheaper(mach1,mach2);
```

T is inferred to be DrinksMachine

- And when we have:

```
ScrabbleWord word3 = PricedObjectOps.cheaper(word1,word2);
```

T is inferred to be ScrabbleWord

- Without this, we would have to cast the return value to the specific type

Comparable

- Java's code library has the interface

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

- The type `Comparable<T>` is used in Java's built-in sorting code etc
- So we must declare e.g.

```
class ScrabbleWord implements Comparable<ScrabbleWord>
```

in order for collections of objects of type `ScrabbleWord` to be sorted by Java's built-in code

Generic interfaces

- If a class implements a non-generic interface, it has to have methods with exactly the same headers as in the interface
- If a class implements a generic interface with the type variable bound to a particular value, it has to have methods with headers like those from the interface except the type variable is replaced by the particular value
- So, for example, a class which implements `Comparable<Thing>` must have within it a method with header

```
public int compareTo(Thing item)
```

Writing our own compareTo

- The method `compareTo` should work so that `obj1.compareTo(obj2)` returns
 - » A negative integer if `obj1` is less than `obj2`
 - » A positive integer if `obj1` is greater than `obj2`
 - » 0 if `obj1` and `obj2` are equal

in the ordering we want for objects of their type

- It is up to us to ensure it works consistently, for example if `obj1.compareTo(obj2)` is negative then `obj2.compareTo(obj1)` should be positive

```
class ScrabbleWord implements Comparable<ScrabbleWord>
{
    private String str;
    private int score;
    public static final int[] scores =
        {1,3,3,2,1,4,2,4,1,8,5,1,3,1,1,3,10,1,1,1,1,4,4,8,4,10};

    public ScrabbleWord(String s)
    {
        str = s;
        String str1 = str.toUpperCase();
        score = 0;
        for(int i=0; i<str1.length(); i++)
            score+=scores[str1.charAt(i)-'A'];
    }

    public String getWord() { return str; }

    public int getScore() { return score; }

    public int compareTo(ScrabbleWord word)
    {
        return word.score-score;
    }

    public String toString() { return str+"("+score+")"; }
}
```

Writing natural order code

If we want to write our own code which is generalised so it can be used for any class of objects which have a natural order, we can do so by writing the code in terms of a type variable `T`, and then declaring the type variable as

```
<T extends Comparable<T>>
```

Example: Selection Sort

```
public static <T extends Comparable<T>> void sort(ArrayList<T> a)
{
    for(int i=0; i<a.size()-1; i++)
    {
        int pos = findMinPos(a,i);
        T temp = a.get(i);
        a.set(i,a.get(pos));
        a.set(pos,temp);
    }
}
```

```
private static <T extends Comparable<T>> int findMinPos
(ArrayList<T> a, int pos)
{
    for(int i=pos+1; i<a.size(); i++)
        if(a.get(i).compareTo(a.get(pos))<0)
            pos=i;
    return pos;
}
```


More bounded types

- `<T extends Thing>` means `T` can be `Thing` or any subclass of `Thing` (upper bound)
- `<T super Thing>` means `T` can be `Thing` or any supertype of `Thing` (lower bound)
- Strictly, generic natural order code needs to be written with a type variable declared as:
`<T extends Comparable<? super T>>`
to cover classes which inherit their `compareTo` method from a superclass
- There much less need to use the `super` bound as opposed to the `extends` bound in generic typing, for the purpose of this module you will never be asked to write complex generic type declarations

Example

```
static <T> void addTo(ArrayList<? extends T> source, ArrayList<T> sink)
{
    for(int i=0; i<source.size(); i++)
        sink.add(source.get(i);
}
```

or

```
static <T> void addTo(ArrayList<T> source, ArrayList<? super T> sink)
{
    for(int i=0; i<source.size(); i++)
        sink.add(source.get(i);
}
```

```
class PositionedScrabbleWord extends ScrabbleWord
{
    private int x,y;
    private boolean horizontal;

    public PositionedScrabbleWord(String s, int x, int y, boolean horiz)
    {
        super(s);
        this.x=x;
        this.y=y;
        horizontal=horiz;
    }

    public int getX() { return x; }
    public int getY() { return y; }
    public boolean getHoriz() { return horizontal; }
}
```

PositionedScrabbleWord inherits compareTo from ScrabbleWord, so it implements Comparable<ScrabbleWord> not Comparable<PositionedScrabbleWord>

Type Variables for Comparison code

- In order to be able to deal with cases like `PositionedScrabbleWord` the type variable declaration for a sort method using objects' own `compareTo` actually has to be

```
public static <T extends Comparable<? super T>>  
void sort(ArrayList<T> a)  
{  
...  
}
```

Java Types

- The Java type system is more complex than a separate type for each class of objects
- An object has several types, the class its constructor is in, and any classes/interfaces that class extends/implements, directly or indirectly
- The type of a variable tells us the methods that can be called on that variable
- The Java interface mechanism enables us to define a type in terms of methods, so we can write code where all we care about is the methods that can be called on objects

Generic Types

- A generic type has to be combined with another type (or more than one) to make a complete type: `GenType<BaseType>`
- If we don't care what the base type is, we can write code where it is set to a type variable
- We can still use a type variable if we want to call methods on objects of the base type, but it must be given a bound of a type containing those methods
`<T extends SomeType> ... GenType<T>`

Why a Duck?

- From a talk given by John Brewer where he discussed the key aspects of object-oriented programming, with an emphasis on polymorphism.
- He defines polymorphism as using objects of different classes interchangeably
- Interface types are how this is done in Java
- Brewer's examples involve an interface type used in some common design patterns

Quackable

```
interface Quackable
{
    void quack();
}
```

- A type for anything which can quack
- A Duck can quack, a Duck Call can quack
- A Duck can `quack()`, a `DuckCall` can `quack()`
- Code where all we require from an object is that it can `quack()` can be written referring to objects as of type `Quackable`

Duck

```
public class Duck implements Quackable
{
    private String myName;
    public Duck(String theName)
    {
        myName = theName;
    }
    public void quack()
    {
        System.out.println(myName+": Quack!");
    }
}
```

DuckCall

```
public class DuckCall implements Quackable
{
    private String myName;
    public DuckCall(String theName)
    {
        myName = theName;
    }
    public void quack()
    {
        System.out.println(myName+": Fake Quack!");
    }
}
```

Polymorphism

```
Quackable quacker;
```

```
quacker = new Duck( "Donald" );  
quacker.quack( );
```

```
quacker = new DuckCall( "Duck Call" );  
quacker.quack( );
```

Factory methods

- A factory method is like a constructor, it returns a new object
- A constructor must return a new object of its actual type
- A factory method may return an object of a subtype of its return type, the code which calls it may not know its actual type

Static factory method

```
public class QuackFactory1
{
    public static Quackable createQuacker(String nm)
    {
        return new DuckCall(nm);
    }
}
```

=====

```
Quackable quacker =
    QuackFactory1.createQuacker("My Duck");
quacker.quack();
```

Decorator

```
public class QuackDecorator implements Quackable
{
    private Quackable myQuackable;
    public QuackDecorator(Quackable theQuackable) {
        myQuackable = theQuackable;
    }
    public void quack() {
        ourQuackCount++;
        myQuackable.quack();
    }
    private static int ourQuackCount = 0;
    public static int getQuackCount() {
        return ourQuackCount;
    }
}
```

Factory Method Revisited

```
public class QuackFactory2
{
    public static Quackable createDuck(String nm)
    {
        return new QuackDecorator(new Duck(nm));
    }

    public static Quackable createDuckCall(String nm)
    {
        return new QuackDecorator(new DuckCall(nm));
    }
}
```

Composite

```
public class QuackComposite implements Quackable
{
    private ArrayList<Quackable> myQuackers =
        new ArrayList<Quackable>();
    public void addQuacker(Quackable quacker)
    {
        myQuackers.add(quacker);
    }
    public void quack()
    {
        for(int i=0; i<myQuackers.size(); i++)
            myQuackers.get(i).quack();
    }
}
```


Goose

```
public class Goose
{
    private String myName;
    public Goose(String name)
    {
        myName = name;
    }
    public void honk()
    {
        System.out.println(myName + ": Honk!");
    }
}
```

Adapter

```
public class GooseAdapter implements Quackable
{
    private Goose myGoose;

    public GooseAdapter(Goose aGoose)
    {
        myGoose = aGoose;
    }

    public void quack()
    {
        myGoose.honk( );
    }
}
```

Factory Object

```
public class QuackFactory3
{
    private String surname;
    public QuackFactory3(String theName)
    {
        surname=theName;
    }
    public void setName(String theName)
    {
        surname=theName;
    }
    public Quackable createDuck(boolean fake,String nm)
    {
        if(fake) return new DuckCall(nm+" "+surname);
        else return new Duck(nm+" "+surname);
    }
}
```

```
QuackFactory3 factory =  
    new QuackFactory3("Smith");  
Quackable quacker1 =  
    factory.createDuck(false, "Donald");  
quacker1.quack();  
Quackable quacker2 =  
    factory.createDuck(true, "Artificial");  
quacker2.quack();  
factory.setName("Jones");  
Quackable quacker3 =  
    factory.createDuck(false, "Daisy");  
quacker3.quack();  
Quackable quacker4 =  
    factory.createDuck(false, "Daniel");  
quacker4.quack();
```

Design Patterns

- Common ways of putting together code
- Provides a vocabulary for programmers
- Not always precisely defined
- Much of the power of design patterns comes from the object-oriented concept of polymorphism
- Using polymorphism can make your code much more flexible