Chapter 10, Interfaces and Inheritance

John M. Morrison

January 19, 2021

Contents

0 Introduction

So far, we have been programming "in the small." We have created simple classes that carry out fairly straightforward chores. Our programs have been little one or two class programs. One class has been the class you are writing, the other has been `jshell` or a simple driver class with a `main` method in it. We created the `BigFraction` API, which allows a client programmer using it to do exact, extended-precision rational arithmetic.

So far, the relationship between classes has been a "has-a relationship." For example our `BigFraction` class has two `BigIntegers`, representing the numerator and denominator of our fraction object. We often use instances of classes that we attach to local variables inside of methods. This is a "uses-a" instead of a "has-a" relationship. Both of these relationships are `compositional`, since we are using them to compose, or build, our class. The compositional relationship is the most important and most common relationship between classes.

Java programs often consist of many classes, which work together to do a job. Sometimes we will create classes from scratch, sometimes we will aggregate various types of objects in a class, and sometimes we will customize existing classes using *inheritance*. We will also draw upon Java's vast class libraries. We will also see how to tie related classes together by using *interfaces*. These are offers to sign a contract whose terms are fulfilled by implementing the methods specified by the interface. Classes accepting this contract are said to *implement* the interface. What is interesting here is that you can create variables of interface type which can point at objects of any class that implements the interface.

Once this machinery is in place, we will look at functional interfaces, which allow us to create function-like objects. These objects are absolutely key to the building of graphical user interface programs.

1 Interfaces

An interface in Java is an offer to sign a contract. This is best seen via an example. We shall create an interface for shapes; for our purposes, a shape is an object that can compute its diameter, perimeter, and area. Here is our

interface. Notice the use of the new keyword `interface`. You can compile this and you will get a `.class` file.

```
public interface Shape
{
    public double area();
    public double perimeter();
    public double diameter();
}
```

For your class to fulfil the terms of the contract, it must implement all of the methods specified in the `Shape` interface. Let us make a `Rectangle` class that implements the three methods. Notice the "`implements Shape`" in the header of the class. This is how we accept the offer made by the interface.

```
public class Rectangle implements Shape
{
    private double width;
    private double height;
    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }
    public Rectangle()
    {
        this(0,0);
    }
    public double area()
    {
        return height*width;
    }
    public double perimeter()
    {
        return 2*(height + width);
    }
    public double diameter()
    {
        return Math.hypot(height, width);
    }
}
```

Next, we create a Circle class. Both these classes implement `Shape`.

```
public class Circle immplements Shape
{
```

```
    private double radius;
    public Circle(double radius)
    {
        this.radius = radius;
    }
    public Circle()
    {
        this(0);
    }
    public double area()
    {
        return Math.PI*radius*radius;
    }
    public double perimeter()
    {
        return 2*Math.PI*radius;
    }
    public double diameter()
    {
        return 2*radius;
    }
}
```

1.1 Pretty Polymorphism

Interfaces, like classes, are types. Since `Circle` and `Rectangle` implement `Shape`, we say that `Circle` and `Rectangle` are *subtypes* of `Shape`, and that `Shape` is a *supertype* of `Circle` and `Rectangle`.

Recall that we have previously said that both varibles and objects have type, and that variables can only point at objects of their own type. Given the tools you have had, this is true. We now reveal a little more about type rules. Let us inspect our class and interface in `jshell` Note that since we are implementing `Shape` we must open it too in the session. This is just what you expect to see.

```
jshell> /open Shape.java

jshell> /open Rectangle.java

jshell> Rectangle r = new Rectangle(6,8)
r ==> Rectangle@685f4c2e

jshell> r.area()
$5 ==> 48.0
```

```
jshell> r.diameter()
$6 ==> 10.0

jshell> r.perimeter()
$7 ==> 28.0
```

Let us continue our session. Now we create a variable of type `Shape`. Since `Rectangle` is a subtype of `Shape`, a variable of type `Shape` can point at a `Rectangle`.

```
jshell> /open Circle.java

jshell> Shape s = new Rectangle(6,8);
s ==> Rectangle@2ef1e4fa

jshell> s.diameter()
$10 ==> 10.0

jshell> s.perimeter()
$11 ==> 28.0

jshell> s.area()
$12 ==> 48.0
```

This works for circles, too.

```
jshell> s = new Circle(10);
s ==> Circle@46f7f36a

jshell> s.diameter()
$14 ==> 20.0

jshell> s.perimeter()
$15 ==> 62.83185307179586

jshell> s.area()
$16 ==> 314.1592653589793
```

Here is what you can't do.

```
jshell> s = new Shape()
|  Error:
|  Shape is abstract; cannot be instantiated
|  s = new Shape()
|      ^---------^
```

Clearly this makes no sense because none of the methods specified in the interface has any code! This is a compile-time error. You cannot create instances of an interface. You can, however, create varibles of interface type. Said variables can point at any object whose type is a subtype of the interface's type. There are two principles at work here.

- **The Visibility Principle** The type of a variable pointing at an object determines what methods are visible; this is true for variables of class or interface type. Only methods in the variable's type may be seen. This is because Java is statically typed; visible methods must be known at compile time.
- **The Delegation Principle** If a variable is pointing at an object and a visible method is called, the object is responsible for executing the method. Regardless of a variable's type, if a given method in the object is visible, the object's method will be called. Remember objects are strongly aware of their type so you can do this.

Programming Exercises

1. Create a class `Triangle.java` that has as state variables the three sides of a triangle. To do this, you need *Herron's Formula*, which goes as follows. Suppose a triangle has sides of lengths a, b, and c. Then its `semiperimeter` s is defined as
$$s = \frac{a+b+c}{2}.$$
. The area of the triangle is
$$\sqrt{s(s-a)(s-b)(s-c)}$$
If the radicand is negative, you have an "illegal" triangle; in that case throw an `IllegalArgumentException`.
2. Make your class `Triangle`, implement `Shape`.
3. Extend `Triangle` to `EqualateralTriangle`, doing as little work as possible.
4. Make `Triangle` implement `Polygon`.
5. Declare and `ArrayList<Shape>` and add shapes to it. Implement this method in a new class's `main`.
```
public static double totalArea(ArrayList<Shape> al)
{
    return 0;
}
```
Create an array list of `Shapes` and test it out.

2 The API Guide

The API guide documents both classes and interfaces. Open the page for `ArrayList`. Near the top of the class, you will see this.

All Implemented Interfaces: `Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

Interfaces, like classes, can have type parameters. What you see listed are all of the standard library interfaces implemented by `ArrayList`.

Click on the link for `List<E>`. Near the top you will see this.

All Known Implementing Classes: `AbstractList, AbstractSequentialList, ArrayList, AttributeList, CopyOnWriteArrayList, LinkedList, RoleList, RoleUnresolvedList, Stack, Vector`

This is a complete list of standard library classes implementing this interface. The Method Summary gives the methods required for a class to implement the interface. You can see that it is a rather large list of methods.

Now look up the interface `Comparable`. It specifies one method, `int compareTo(T o)`. You will see that there are swarms of classes that implement this interface, including `String` and `BigInteger`. A comparable type consists of sortable data. Implmenting this interfaces turns on some very valuable sorting tools built into Java.

3 Classes and Subtypes

You might ask, "Can one class be a subtype of another?" The answer is yes, and the purpose of this section is to show you how to do it. The mechanism we shall use is called *inheritance*. Again, we will illustrate this with an example. It would seem that a square is a special type of rectangle, so if we create a class `Square` it should be a subtype of `Rectangle`. The `extends` keyword indicates that `Square` is subtype of `Rectangle`. Automatically, `Square` gets all of `Rectangle`'s public methods. However, in this state we do not have an appropriate constructor.

```
public class Square extends Rectangle
{
}
```

Note that to compute arae, diameter, and perimeter of a square, we just need to know its side length. So we we begin by adding a state variable for that.

```
public class Square extends Rectangle
{
```

```
    private double side;
}
```

Now we need a constructor. Here we introduce the keyword `super`, which calls the parent constructor.

```
public class Square extends Rectangle
{
    private double side;
    public Square(double side)
    {
        super(side, side);
        this.side = side;
    }
}
```

Warning! If you use the `super` keyword in a constructor, it must be done in the first line of the constructor's code. Failure to do this is a compile time error. You should reverse the order of the two lines of code in `Square`'s constructor and see what this error looks like.

Now, It's time for inspection.

```
jshell> /open Shape.java

jshell> /open Rectangle.java

jshell> /open Square.java

jshell> Rectangle r = new Square(10);
r ==> Square@2e5d6d97

jshell> r.area()
$5 ==> 100.0

jshell> r.diameter()
$6 ==> 14.142135623730951

jshell> r.perimeter()
$7 ==> 40.0

jshell> Shape s = new Square(10)
s ==> Square@2ef1e4fa

jshell> s.area()
```

```
$9 ==> 100.0

jshell> s.diameter()
$10 ==> 14.142135623730951

jshell> s.perimeter()
$11 ==> 40.0
```

Notice that a subtype of a subtype is a subtype! Happily, we were able to inherit all of `Rectangle`'s methods and making `Square` was easy. Also, note that every type is a subtype of the root class `Object`. Here is a way to convince you.

```
jshell> Object o = "some string"
o ==> "some string"

jshell> Object p = new ArrayList<String>();
p ==> []

jshell> o.toString()
$3 ==> "some string"

jshell> p.toString()
$4 ==> "[]"
```

An `Object` has a `toString()` method, so by the Visibility Principle, an `Object` variable can see a `toString()` method. That is why everything works here.

Now see this.

```
jshell> o.length()
|  Error:
|  cannot find symbol
|    symbol:   method length()
|  o.length()
|  ^------^
```

Strings have a `length()` method, but `Objects` do not. The error you see is caused by the fact that an `Object` variable can only see `Object` methods; hence this compile-time error. A similar thing happens when we try to get the `ArrayList`'s size.

```
jshell> p.size()
|  Error:
|  cannot find symbol
|    symbol:   method size()
|  p.size()
|  ^----^
```

4 Overriding Methods

Let us now put a `toString` method in our `Rectangle` and `Circle` classes. Insert these into your classes. As it stands now, the `Object toString()` method is just the class name followed by an and some hex digits. Add these to your classes.

```
public String toString()
{
    return String.format("Rectangle(%s, %s)", width, height);
}

public String toString()
{
    return String.format("Circle(%s)", radius);
}
```

```
jshell> /open Shape.java

jshell> /open Circle.java

jshell> /open Square.java

jshell> /open Rectangle.java

jshell> Shape r = new Rectangle(6,8);
r ==> Rectangle(6.0, 8.0)

jshell> Shape s = new Circle(10)
s ==> Circle(10.0)

jshell> Shape sq = new Square(10)
sq ==> Rectangle(10.0, 10.0)
```

We have successfully overriden the `Object toString()` method. When you do this, it is a smart idea to use the `@Override` annotation on these methods. Here is what it looks like on `Rectangle`.

```
@Override
public String toString()
{
    return String.format("Rectangle(%s, %s)", width, height);
}
```

and here it is cor `Circle`.

```
public String toString()
{
    return String.format("Circle(%s)", radius);
}
```

Use this annotation whenever overriding the method of an ancestor class. This causes the compiler to check to see if you are overriding properly. One thing that can happen is that you use the wrong signature in the method you are intending to override. In this case, you don't override the method, you *overload* it with method name overloading. This can cause unfortunate and hard to diagnose errors.

One thing we see that is unsatisfactory is the `Rectangle` `toString()` method is not really right thing for `Square`. We fix this by overriding the parent method. Add this to your `Square` class and you will like the result (Run it!).

```
@Override
public String toString()
{
    return String.format("Square(%s)", side);
}
```

By now you have likely surmised this. If you call a method on a object an that method is implemented in the object's class, that method is called. If the method is not present, then the JVM checks for it in the parent class and runs it if it is present there. This process will continue all the way up to the root class `Object`. If it's not found by then, the compiler will issue forth with an error message.

Programming Exercises

1. Make these classes, `Mama.java`

   ```
   public class Mama
   {
       public int foo()
       {
           return 42;
       }
   }
   ```

   and `Baby.java`

   ```
   public class Baby extends Mama
   {
       public int foo(int x)
       {
   ```

```
            return 2*x;
        }
    }
```

2. Compile them.
3. Make this clas

```
public class Main
{
    public static void main(String[] args)
    {
        Mama m = new Mama();
        Baby b = new Baby();
        System.out.println(m.foo());
        System.out.println(b.foo());
        System.out.println(b.foo(10));
    }
}
\item Explain what happened.
\item Insert an \texttt{@Override} annotation on \texttt{foo} in
    the class \texttt{Baby}.  Compile. What happened?
```

5 Inheritance and the API Guide

Open the page for `ArrayList`. Right at the top of the page you will see this.

Module `java.base`

Package `java.util`

Class `ArrayList<E>`

```
java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>
```

Type Parameters: E - the type of elements in this list

All Implemented Interfaces:

`Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess`

Direct Known Subclasses:

`AttributeList, RoleList, RoleUnresolvedList`

You can see the "family tree" of `ArrayList` displayed going all the way back up to `Object`. You can also see the child classes of `ArrayList` in the standard libraries.

Now scroll past the method summary. You will see all of the methods inherited from ancestor classes. The method names are links; click on them to see their method detail.

Exercises These will help you to learn to use the documentation more effectively.

1. From which class does `ArrayList` inherit its `toString()` method?
2. From which class does `ArrayList` inherit its `sort()` method?
3. From which class does `ArrayList` inherit its `equals()` method?
4. From which class does `ArrayList` inherit its `wait()` method?
5. Whose `add` methood is overriden by `ArrayList`?

6 Extending Interfaces

Interfaces can be extended to require additional methods in implementing classes. For example, we could extend our `Shape` interface as follows.

```
public interface Polygon extends Shape
{
    public int numSides();
}
```

An implementing class for this interface must implement all shape methods and implement `numSides()` as well. Let us do this to `Rectangle`.

```
public class Rectangle implements Polygon
{
    private double width;
    private double height;
```

```
    public Rectangle(double width, double height)
    {
        this.width = width;
        this.height = height;
    }
    public Rectangle()
    {
        this(0,0);
    }
    public double area()
    {
        return height*width;
    }
    public double perimeter()
    {
        return 2*(height + width);
    }
    public double diameter()
    {
        return Math.hypot(height, width);
    }
    @Override
    public String toString()
    {
        return String.format("Rectangle(%s, %s)", width, height);
    }
    public int numSides()
    {
        return 4;
    }
}
```

Notice that there is no suitable `numSides()` method for a `Circle`, so we only implement `Shape` in that case. Observe that `Square` implements `Polygon` automatically since implemented interfaces are inherited. There is no need to override here, since squares and rectangles both have four sides. Also note this; a variable of `Shape` type cannot see `numSides`. For this, you need a variable of type `Polygon`.

However, `Square` will inherit the implementation of `Polygon` from `Rectangle`.

Programming Exercises

1. What are the superinterfaces of `java.util.Collection<E>`?
2. Is `ArrayList<E>` a subtype of `java.util.Collection<E>`?

3. Of what interface(s) is `List<E>` a direct subinterface?

6.1 Default Methods

Beginning in Java8, methods called `default methods` were allowed to be placed into interfaces. Here is how the work.

If you implement an interface with a default method, you automatically "inherit" the default method. However, if you don't like what that method does, you are free to override it by implementing the method in your class.

If you implement two interfaces that have the same default method, you *must* override that method to avoid the deadly diamond problem. This convention is enforced by Mean Mr. Compiler.

7 Functional Interfaces

An interface is a *functional interface* if it specifies exactly one method. These interfaces can have default methods as exceptions to this rule but they are the only exceptions.

We will begin by studying the `Consumer` family of interfaces. This consists of four interfaces, all of which live in package `java.util.function`.

- `Consumer<T>` This is a generic interface and specifies a function taking an object of type `T` as input and having a void return type.
- `IntConsumer` This is interface and specifies a function taking an `int` as input and having a void return type.
- `LongConsumer` This is an interface that specifies a function taking an `Long` as input and having a void return type.
- `DoubleConsumer` This is an interface that specifies a function taking an `double` as input and having a void return type.

In all cases, the name of the method is `accept`. If you look in the API Guide, you will see the default method `default Consumer<T> andThen(Consumer<? super T> after`. Since this is a default method, we do not need to worry about or implement it now. Consumers take a piece of data and do something with it, such as printing it to `stdout`.

If you want to implement this, you might at first think, "Crud. I have to make a class that implements the desired consumer, put an `accept` method in it, and then create an instancd of of it to use it. Ugh. This is utterly useless, right?

Wrong. A fiendlishly clever mechanism of type inference makes using these interfaces simple. First, realize that you can create a variable of interface type. Secondly, Java has two means of creating function-like objects. First, let us introduce the *method reference*. Bear witness to this snippet of `jshell`.

```
jshell> Consumer<String> printMe = System.out::println;
printMe ==> $Lambda$19/0x0000000800b5a440@17f052a3

jshell>
```

What in tarnation? Why is this allowed? There is a friendly ghost here; it is a class with no name. We have created an object that is an instance of a nameless class that is a subtype of `Consumer<String>`. You now ask, "This is a pretty new toy, but what can you do with it.?"

It's time for a trip to the `ArrayList` class. Look at the method detail for the `forEach` method.

```
public void forEach(Consumer<?  super E> action)
```

Description copied from interface: `Iterable` Performs the given action for each element of the Iterable until all elements have been processed or the action throws an exception. Actions are performed in the order of iteration, if that order is specified. Exceptions thrown by the action are relayed to the caller. The behavior of this method is unspecified if the action performs side-effects that modify the underlying source of elements, unless an overriding class has specified a concurrent modification policy.

Specified by:

`forEach in interface Iterable<E>`

Parameters:

`action` - The action to be performed for each element

Throws:

`NullPointerException` - if the specified action is null

You see some terrifying notation here, this `<?  super E>` thing. This just means that the `forEach` method takes a `Consumer` of any type that is a super-type of the type populating the array list. In particular, our `Consumer` will take any `Consumer` of `String`s. Now watch this.

```
jshell> ArrayList<String> al = new ArrayList<>();
al ==> []

jshell> al.add("aardvark")
$4 ==> true
```

```
jshell> al.add("bear");
$5 ==> true

jshell> al.add("carical");
$6 ==> true

jshell> al.forEach(printMe)
aardvark
bear
carical

jshell> al.forEach(System.out::println)
aardvark
bear
carical
```

What's cool is this `printMe` variable can store a *method*. This eliminates the need for writing a loop to display the items in a list. but wait... there is more!

8 Lambdas

Is there a way to customize the function object we want to create that is more flexible than a method reference? It turns out there is; these objects are called *lambdas*, and they are functions without a name. It is easy to make a lambda that is a Consumer. Here is a very simple example.

```
jshell> Consumer<String> kappa = e -> System.out.println(e.toUpperCase());
kappa ==> $Lambda$21/0x0000000800b5c440@6433a2

jshell> al.forEach(kappa)
AARDVARK
BEAR
CARICAL
```

Java is doing a little type inference here; it knows that `kappa` is a `Consumer<String>`, so it infers that the argument `e` is a `String`. The expresson on the right-hand side of the arrow is the return value for the lambda.

The `forEach` method takes a consumer, which basically is an action, and performs it on each item in list in index order.

8.1 Lambda Grammar

Lambdas can come in a variety of forms. We shall present a field guide here. The simplest form looks like this for a function with one argument and a return value.

```
x -> expr
```

The item `expr` is the return value of the lambda. If you have two arguments, you must enclose them in parentheses and your lambda looks like this.

```
(x, y) -> expr
```

You can enforce a masure of type safety by specifying types for arguments; here is an example of that.

```
(String s, int y) -> s.charAt(y)
```

You can have lambdas with more than one line of code. If you want to return something from such a lambda, you must do so explicitly. Here is a simple example that computes the square root of the sum of two numbers.

```
(int x, int y) ->
{
    int z = x > 0? x: -x;
    z += y > 0? y: -y;
    return z;
}
```

9 Comparators and Sorting

Often we have a collection of objects that are sortable in various ways and we would like a mechanism that would conveniently afford us this ability. To this end, we now undertake a brief study of the functional interface `Comparator<E>`.

```
public interface Comparator<E>
{
    public int compare(E e1, E e2);
}
```

The design contract works like this. The set of elements `E` is given an ordering $\leq$ by this `compare` method. This ordering works as follows.

$$\texttt{e1} \leq \texttt{e2} \iff \texttt{compare}(\texttt{e1}, \texttt{e2}) \leq 0.$$

This ordering should satisfy these properties. For all `e1`, `e2` and `e3`,

1. If e1.equals(e2), then compare(e1, e2) returns 0.
2. if e1 $\leq$ e2 and e2 $\leq$ e3, e1 $\leq$ e3 (transitivity).
3. (e1 $\leq$ e2) $\vee$ (e1 $\leq$ e2) is always true. In other words, any two elements of E are "related." Mathematically, this ordering is said to be *linear*. This is exactly what we expect for objects to be deemed sortable.

We will do a modest example here, which will allow us to sort an entry from a concordance by frequency of appearance and secondarily alphabetically, or alphabetically and then by frequency. We begin by showing the basic class.

```
public class ConcordanceEntry
{
    String word;
    int times;
    public ConcordanceEntry(String word, int times)
    {
        this.word = word;
        this.times = times;
    }
    @Override
    public String toString()
    {
        return String.format("%s:  %s", word, times);
    }
}
```

Next, we add some two static elements of type Comparator; notice that these variable are of interface type. We have made them final because they are exposed to the client.

```
public class ConcordanceEntry
{
    public static final Comparator<ConcordanceEntry> byTimes;
    public static final Comparator<ConcordanceEntry> byFrequency
    static
    {
        byTimes = (e1, e2) -> {
            if(e1.times != e2.times)
            {
                return e1.times - e2.times;
            }
            return e1.word.compareTo(e2.word);
        };
        byAlpha = (e1, e2) -> {
```

```
            if(!e1.word.equals(e2.times))
            {
                return e1.word.compareTo(e2.word);
            }
            return e1.times - e2.times;
        };
    }
    String word;
    int times;

    public ConcordanceEntry(String word, int times)
    {
        this.word = word;
        this.times = times;
    }
    @Override
    public String toString()
    {
        return String.format("%s:  %s", word, times);
    }
}
```

What is happening here? How are these two new static members actual objects?

If you visit the API guide, you will see that the interface `Comparator` is a functional interface. It specifies one method,

```
public int compare(T e1, T e2);
```

Java is performing type inference here we described in the our discussion of `Consumers`. It creates an object of class type that implements the specified method and it assigns the lambdas we assigned to the static variables as the `compare` method of that object. No specific named class is ever created.

In pre-8 Java, you would need to implement these using an anonymous inner class. The code for byAlpha would look like this.

```
        byAlpha = new Comparator<ConcordanceEntry>(){
            public int compareTo(Concordance e1,
                ConcordanceEntry e2){
            if(!e1.times.equals(e2.times))
            {
                return e1.word.compareTo(e2);
            }
            return e1.times - e2.times;
        };
```

This new construct of lambdas eliminates a good bit of boilerplate code that adds no meaning to what we are doing. What we want to pass here is *behavior* that is to be carried out by our comparator objects.

So what is the benefit of this? How do we sort? Let us demonstrate that in a main method. To implement this method, we introduce the static method `sort` in the class `java.util.Collections`. There are two methods by this name. We use the `sort` method which has the signature `[List<E>, Comparator<E>]`; this method sorts according to our comparator.

```
public static void main(String[] args)
{
    ArrayList<ConcordanceEntry> al =
        new ArrayList<>();
    al.add(new ConcordanceEntry("cow", 5));
    al.add(new ConcordanceEntry("pig", 2));
    al.add(new ConcordanceEntry("zebra", 3));
    al.add(new ConcordanceEntry("cow", 5));
    al.add(new ConcordanceEntry("zebra", 5));
    al.add(new ConcordanceEntry("elephant", 6));
    al.add(new ConcordanceEntry("eland", 1));
    al.add(new ConcordanceEntry("coati", 2));
    System.out.println("Unsorted:");
    for(ConcordanceEntry e: al)
    {
        System.out.println(e);
    }
    Collections.sort(al, byAlpha);
    System.out.println("Sorted by word:");
    for(ConcordanceEntry e: al)
    {
        System.out.println(e);
    }
    System.out.println("Sorted by frequency:");
    Collections.sort(al, byTimes);
    for(ConcordanceEntry e: al)
    {
        System.out.println(e);
    }
}
```

Running this, here is the original list.

```
cow:  5
pig:  2
zebra:  3
```

```
cow:  5
zebra:  5
elephant:  6
eland:  1
coati:  2
```

Here it is, sorted first by word and second by frequency.

```
Sorted by word:
eland:  1
coati:  2
pig:  2
zebra:  3
cow:  5
cow:  5
zebra:  5
elephant:  6
```

Now we sort first by frequency then by word.

```
eland:  1
coati:  2
pig:  2
zebra:  3
cow:  5
cow:  5
zebra:  5
elephant:  6
```

10 Can I have many parents?

One question you might ask is, "Can I implement more than one interface?" The answer is yes. You do this.

```
public class ImplementingClass implements Interface1,
    Interface2, .... InterfaceN
{
    //code
}
```

When you do this, your class, or one of its ancestors, must implement the totality of all of the methods specified in the interface. This fact has an appealing and useful feel.

10.1 The Deadly Diamond

The next queston you might ask is "Can I have several parent classes?" The answer here is no. Class designers often speak of the "deadly diamond;" this is a big shortcoming of multiple inheritance and can cause it to produce strange behaviors. Imagine you have these four classes, Root, Left, Right and Bottom. Suppose that `Left` and `Right` extend `Root` and that `Bottom` were allowed to extend `Left` and `Right`.

Before proceeding, draw yourself a little inheritance diagram. Graphically these four classes create a cycle in the inheritance graph (*which in Java must be a rooted tree*).

Next, imagine that both the `Left` and `Right` classes implement a method `f` with identical signature and return type. Further, suppose that `Bottom` does not have its own version of `f`; it just decides to inherit it. Now imagine seeing this code fragment

```
Bottom b = new Bottom(....);
b.f(...)
```

There is a sticky problem here: Do we call the `f` defined in the class `Left` or `Right`? If there is a conflict between these methods, the call is not well–defined in our scheme of inheritance.

10.2 A C++ Interlude

There is a famous example of multiple inheritance at work in C++. There is a class `ios`, with children `istream` and `ostream`. The familiar `iostream` class inherits from both `istream` and `ostream`. Since the methods for input and output do not overlap this works well.

However, the abuse of multiple inheritance in C++ has lead to a lot of very bad errors in code. Java's creators decided this advantage was outweighed by the error vulnerabilities of multiple inheritance.

The One-Parent Rule Every class has exactly one parent, except for `Object`, which is the root class. When you inherit from a class, you "blow your inheritance." The ability to inherit is very valuable, so we should only inherit when it yields significant benefits.

11 Abstract Classes

Interfaces do some nifty work for us. They afford us polymorphism which gives some desirable flexibility. They allow a project manager to specify names and signatures of methods for his programmers to develop. You can even insert default methods into interfaces. You cannot, however have a constructor for an interface, nor can you give an interface state. Interfaces also enjoy the advanage that a class can implement several of them, where you are always limited to one parent class via inheritance.

Sometimes, a group of related classes have a lot in common and you want to share common state and common code. This is in accordance with the 11th Commandment: *Thou shalt not maintain duplicate code.*

Fortunately there is a construct that allows us to "partially implement" a class, prevent the class from being instantiated, and force the children of the class to implement methods in a manner similar to that of an interface. Such a beast is called an `abstract class`.

In anticipation of the future, let us think again about shapes, but this time we will create a group of shape classes that can draw themselves in some kind of graphical environment. A shape that is going to draw itself needs to know its center, its color, and its size. What is different about various shape types is how they draw themselves.

For now, we will produce some "toy" classes as a means of demonstration; in a subsequent chapter we will use the scheme we create here to build a GUI application that renders shapes on a graphics surface.

Create these classes. Here is `Color.java`

```
public class Color
{
    private final int hexCode;
    public Color(int hexCode)
    {
        this.hexCode = hexCode;
    }
    @Override
    public String toString()
    {
        return String.format("Color(#%s)", Integer.toString(hexCode, 16));
    }
}
```

Here is `Canvas.java`

```
public class Canvas
```

```
{
    private final Pen pen;
    private double width;
    private double height;
    public Canvas(double width, double height)
    {
        this.width = width;
        this.height = height;
        this.pen = new Pen();
    }
    public Pen getPen()
    {
        return pen;
    }
    public double getWidth()
    {
        return width;
    }
    public double getHeight()
    {
        return height;
    }
    @Override
    public String toString()
    {
        return String.format("Canvas(%s, %s)", width, height);
    }
}
```

Here is `Pen.java`.

```
public class Pen
{
    private Color color;
    private double width;
    private void setColor(Color color)
    {
        this.color = color;
    }
    private void setWidth(double width)
    {
        this.width = width;
    }
    @Override
    public String toString()
    {
```

```
        return String.format("Pen(color = %s, width = %s)", color, width);
    }
}
```

Now let us make a class for a shape, `Shape.java`.

```
public class Shape
{
    private double centerX;
    private double centerY;
    private double color;
    private double size
    public Shape(double centerX, double centerY, double color, double size)
    {
        this.centerX = centerX;
        this.centerY = centerY;
        this.size = size;
    }
    public void draw(Pen pen)
    {
        //what do we do here?
    }
```

We see a problem: how do we draw a general `Shape`? That makes no sense whatsoever! This is a use case for abstract classes. Java has a keyword `abstract` that can be applied to classes and methods. We will leave the `draw` method undefined just as we would in an interface. To do this, we must declare the method `abstract`, like so

```
public abstract draw(Pen pen);
```

If we mark any method in a class `abstract`, we must also mark the class `abstract`. Our class now looks like this.

```
public abstract class Shape
{
    private double centerX;
    private double centerY;
    private double color;
    private double size
    public Shape(double centerX, double centerY, double color, double size)
    {
        this.centerX = centerX;
        this.centerY = centerY;
        this.size = size;
```

```
}
public abstract void draw(Pen pen);
```

Here are the rules of the road for abstract classes.

1. If you omit a method's body in a class you *must* declare that method `abstract`.
2. If *any* method in a class is declared `abstract`, the class itself must be declared `abstract`.
3. You may declare any class you create `abstract`. By so doing, you prevent any instances of it from being created; do this if it make no sense for an instance of your class to be created.
4. You may not create instances of any abstract class.
5. You can create variables of abstract class type. They can point at any object of any descendant type. Both the visibility and delegation principles apply.
6. If you extend an abstract class and the child class is not abstract, you must implement all abstract methods in the abstract class, as well as any abstract methods in ancestor classes of the abstract class.
7. A class that is not abstract is said to be `concrete`.

A variable of type `Shape` can point at any object of a descendant type. You could now make classes for circles, rectangles, squares, and other graphical objects. These can each implement the `draw` method of `Shape`.

How do I know if a standard library class is abstract? Look in the Java API guide and find the class `AbstractList`; clearly it will be abstract. Go to the top of the page. You will see the fully-qualified name, the family tree, and then its implemented interfaces and direct descendants. Just below that you see this

```
public abstract class AbstractList<E>
extends AbstractCollection<E>
implements List<E>
```

The first line tells all: See the word `abstract`?

So, in summary, you can declare any class abstract and instances of it cannot be created. You can declare methods in a class abstract and they cannot have a method body. Any child class must override these methods unless, it too, is abstract. Any class containing an abstract method must be marked abstract. However, an abstract class is not required to have any abstract methods.

sectionA Brief Trip into Functional Programming If you look in the API guide for `ArrayList`, you will find a mysterious method called `stream`. This is among the default methods inherited from the interface `java.util.Collection`. Here is its method detail.

```
default Stream<E> stream()
```

Returns a sequential Stream with this collection as its source.

This method should be overridden when the spliterator() method cannot return a spliterator that is IMMUTABLE, CONCURRENT, or late-binding. (See spliterator() for details.)

Implementation Requirements: The default implementation creates a sequential Stream from the collection's Spliterator.

Returns: a sequential Stream over the elements in this collection

Since: 1.8

So, what is a `Stream<E>`? A stream is a read-only Netflix-like view of a collection of objects. So for an `ArrayList`, a stream containing the list's objects can be created and they are shown in the list's order. Creating a stream creates a `source`. Creating the stream causes little or nothing to happen. You must then take some action on the stream to consume it; consumption of a stream triggers the actual processing of the data in the list.

Here are some methods that trigger consumption.

- `forEach(Consumer<? super E> action` This works just as it does on an array list.
- `count()` This counts the number of elements in the stream and returns the count as an integer.
- `collect(Collectors.toList())` will collect the contents of the stream into a list. returns the count as an integer.

Wait a minute! This seems kind of useless! In its present form, admittedly so. However, we have only described the bread in the sandwich (a source and something that consues), but not the goodness inside. That's where this becomes a very powerful tool.

The goodies inside consist of *transformers* that transform and filter the items in your collection. Transformers are basically alimenatary canals that "eat" from streams and subsequently "excrete" a stream. Let us now meet the `Predicate`

family. They reside in package `java.util.function`. They are all functional interfaces, so we can use lambdas where called for.

- `Predicate<T>` Objects of this type can be specified using a lambda whose argument is of type `T` and whose return value is a `boolean`.
- `IntPredicate`Objects of this type can be specified using a lambda whose argument is of type `int` and whose return value is a `boolean`.
- `LongPredicate`Objects of this type can be specified using a lambda whose argument is of type `long` and whose return value is a `boolean`.
- `DoublePredicate` Objects of this type can be specified using a lambda whose argument is of type `double` and whose return value is a `boolean`.

Let us see `filter` at work.

```
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
public class Filter
{
    public static void main(String[] args)
    {
        ArrayList<String> words = new ArrayList<>();
        words.add("aardvark");
        words.add("African gazelle");
        words.add("ostrich");
        words.add("yak");
        words.add("zebra");
        words.add("vampire bat");
        words.add("anaconda");
        words.add("tapir");
        System.out.println(words);
        words.stream()
             .filter( s -> s.compareToIgnoreCase("m") < 0)
             .forEach(System.out::println);
        List<String> out = words.stream()
                                .filter( s -> s.compareToIgnoreCase("m") < 0)
                                .collect(Collectors.toList());
        System.out.println(out);
     }
}
```

Let us point out some features of this little program. Notice the typographical convention of aligning "on the dots" where we first make a stream from the `ArrayList words`. Note that only the last line in the progression ends with a

semicolon. Actually, this is a single line of code, but we do not want great long lines of code because they become unreadable. Combining a filter with a foreach statement eliminates the need for creating a for loop with an if statement inside for filtering a list.

11.1 Declarative vs. Imperative

The streams API and the functional programming interface in Java enable a style of programming that is *declarative*; to wit, you get exactly what you want by saying what you want. The line

```
words.stream()
     .filter( s -> s.compareToIgnoreCase("m") < 0)
     .forEach(System.out::println);
```

says, "Make a stream from my list, filter in the first half of the alphabet, and print the entries out. The imperative form looks like this.

```
for (String s: words)
{
    if( x.compareToIgnoreCase("m") < 0)
    {
        System.out.println(s);
    }
}
```

Here we are "spoon-feeding" Java every minute step rather than telling it what we want.

11.2 Using `map`

This is the bread-and-butter means for making a stream of objects of one type into a stream of objects as another or the same type. Let's see how we can use it to get our animal list to print upper-case. Notice we have two transformers in our stream sandwich.

```
words.stream()
     .filter( s -> s.compareToIgnoreCase("m") < 0)
     .map( s -> s.toUpperCase())
     .forEach(System.out::println);
```

The output looks like this.

```
AARDVARK
AFRICAN GAZELLE
ANACONDA
```

To do this imperatively, you might do this.

```
for (String s: words)
{
    if( x.compareToIgnoreCase("m") < 0)
    {
        System.out.println(s.toUpperCase());
    }
}
```

Map, like its friend filter, comes in various guises. Here they come.

1. `mapToInt` This changes an object stream into a stream of (primitive) integers, an `IntStream`.
2. `mapToDouble` This changes an object stream into a stream of (primitive) doubles, a `DoubleStream`.
3. `mapToLong` This changes an object stream into a stream of (primitive) longs, an `LongStream`.