Interfaces as Types

Learning Outcomes:

After doing this exercise, the learner…

1. **has been confronted with, and has seen, that an interface can be used as the type when declaring a reference; similarly, it can "refer to" any object ("instance") that is an implementer of the interface.**
2. **has been confronted with the common novice-programmer error of trying to assign weaker privileges to methods when trying to implement an interface.**

Note:

Feel free to experiment and ask questions to the AllQuestionsForum when you come up against something that confuses you.

To Do:

Follow below. Copy and compile the codes yourself and experiment briefly whenever you are curious (oh!, and ask questions).

Part I – Using interfaces as Types

Below, we see a change to the previously shown code where the class `Bird` has been changed to be an interface instead of a class. On the right, we'll see that this required a small change to the `BlackBird` class (see commented lines).
[Note: the data-members `String type` and `String name` (previously declared the `Bird` class) are no-longer declared in the interface Bird - so the `BlackBird` constructor could no longer refer to them; i.e. they weren't inherited.]

BirdMaker.java | Bird.java

```
1  interface Bird
2  {
3      void sing();
4      boolean sings();
5
6  }
7
```

BlackBird.java

```
1  //class BlackBird extends Bird
2  class BlackBird implements Bird
3  {
4      BlackBird()
5      {
6          //type = "blackbird"; //won't compile: no longer "inherited"
7          System.out.println("Blackbird Constructor");
8      }
9
10     void sing()
11     {
12         System.out.println("Blackbird sing()");
13     }
14
15     boolean sings()
16     {
17         return true;
18     }
19 }
```

However, following this minor change – you will see that you can re-compile `Bird`, and BlackBird (the changed source-code) and that `BirdMaker` will work without any changes…

Well, nearly! (Try the two compile statements below and observe the error you get). The next page shows how to fix this.

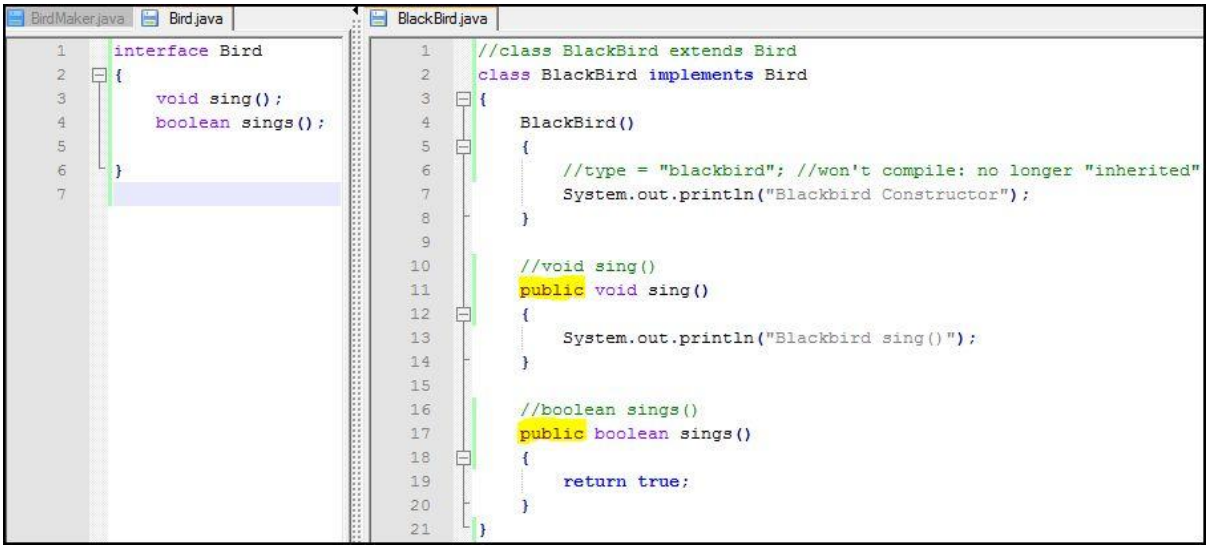
```
G:\example>javac Bird.java

G:\example>javac BirdMaker.java
.\BlackBird.java:15:  error:  sings()  in  BlackBird  cannot  implement
sings() in Bird
        boolean sings()
                ^
  attempting to assign weaker access privileges; was public
.\BlackBird.java:10: error: sing() in BlackBird cannot implement sing()
in Bird
        void sing()
             ^
  attempting to assign weaker access privileges; was public
2 errors

G:\example>
```

Interfaces as Types

The problem above occurred because, in an interface, everything is implicitly public: i.e. even though we wrote: `void sing()` the compiler wrote **`public`** `void sing()`.

There is a rule in Java that if something is public in a super-class or an interface it has to be public in a sub-class or an implementing class.

So: the following fixes it:



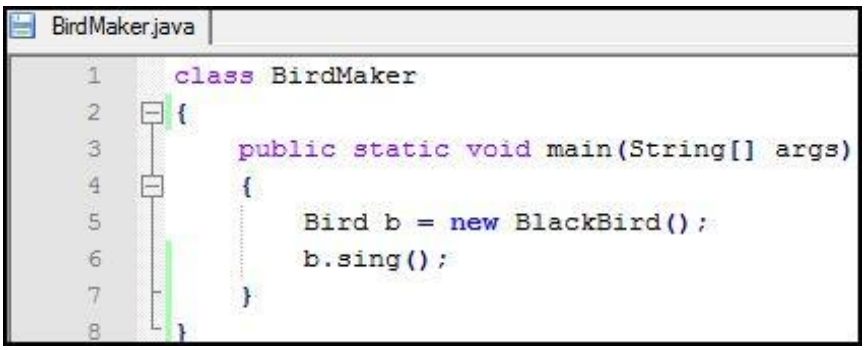
That is, to "implement" an interface's declared methods – you have to put 'public' in the implementing class (as above).

Ok, that done – you save your changes to `BlackBird.java`, go to the compiler, and …

```
G:\example>javac BlackBird.java

G:\example>
```

It compiles!

Now – will BirdMaker work?



Let's see…

```
G:\example>java BirdMaker
Blackbird Constructor
Blackbird sing()
```

It does!

Ok. The lesson here is that you can use an interface as the 'type' of a reference variable – and this means that it can be assigned to any instance that is an implementer of the type.

With a little thought – this makes sense. If you write a class BlackBird and you say:
`class BlackBird implements Bird{…`

you are stating that you are going to implement all methods listed in the `Bird` interface. Not only that – if you don't, the compiler won't compile `BlackBird`. So, from once it compiles, it is guaranteed to have a method implementation for
`public void sing()`
and
`public boolean sings()`
and so it makes sense that a reference variable of type `Bird` – can hold an identifier (or loosely-speaking an ~address) to an object-instance that implements the `Bird` interface.

Lesson Applied

See the code below:

BirdMaker.java

```
1  //please ignore the compiler warning you get
2  //when you compile this code
3  import java.util.List;              //an interface
4  import java.util.ArrayList;         //an implementer of the interfae
5  class BirdMaker
6  {
7      public static void main(String[] args)
8      {
9          List listOfBirds; //reference type is java.util.List (interface)
10         listOfBirds = new ArrayList(); //ArrayList implements java.util.List
11
12         //below: can call 'add( )' method defined in java.util.List
13         // because java.util.ArrayList implements java.util.List
14         listOfBirds.add( new BlackBird() );
15
16         for(Object o: listOfBirds) //for each Object say 'o', in the listOfBirds
17         {
18             ((Bird)o).sing();       //downcast from 'Object' to 'Bird' and
19                                     // call the sing() method
20         }
21     }
22 }
```

Above at line 9, a reference-variable `listOfBirds` is declared with the type List (see import of `java.util.List`) – this is an interface type.

Then at line 10 the reference is assigned a value that "refers to" a newly created `ArrayList` object-instance (see `import java.util.ArrayList;`) – this is a concrete class that implements the `List` interface.

At line 14, the `listOfBirds` reference is used to call the `add(Object o)` method of the concrete `ArrayList` instance which the `listOfBirds` reference-variable is now holding the (~loosely-speaking) address to.

At lines 16 to 20, we loop through the `ArrayList` instances collection of `Bird` objects and use a downcast to access the `sing()` method. [**Note:** *this is a pre-Java-1.5 way of doing this – we should be using* `new ArrayList<Bird>()` *but for the above example we kept it simple.*]

END LAB