16 INTERFACES

Terry Marris 31 May 2001

16.1 OBJECTIVES

By the end of this lesson the student should be able to

- explain what an interface is
- write and implement an interface
- change an instance of one interface implementation to an instance of another implementation of the same interface

16.2 PRE-REQUISITES

The student should be comfortable with Chapter 15, Inheritance.

16.3 PREVIEW

In Java, a class may inherit from just one other class. But a class can implement several interfaces. An interface is essentially a set of operation or method signatures; a class with no implementation. We see how to write an interface. We see how to write classes that implement an interface. We see how several classes can share the same interface. We see that a class can implement several different interfaces - a form of multiple inheritance.

16.4 INTRODUCTION

Interfaces are used extensively in implementing data structures and graphical user interfaces (see Parts Two and Three of the Java Notes).

16.5 CONSISTENCY

Interfaces help ensure consistency between classes that make up an application.

We recall the class diagram modelling a lending library system described in Chapter One (Figure 18.1). Suppose we intend that each object is to be stored in some collection e.g. a set (more usually, the objects would be stored on a file, but we have not come to files yet). We want each class to implement a method that stores its instance. We want each class to use the same signature (perhaps because a different programmer is writing each class). So we create an interface named *Storable* that each object must implement.

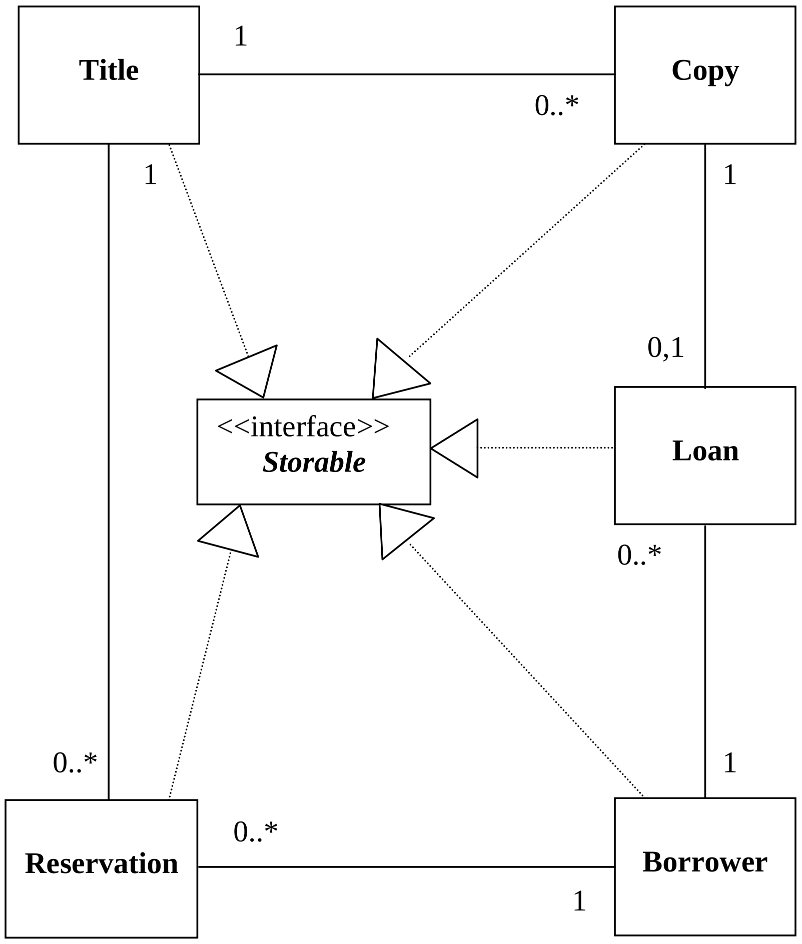

Figure 16.1 *Class Diagram Modelling a Lending Library*

The dotted arrows mean that the class is obliged to implement the interface. To illustrate the principles we shall show how *Copy* implements the *Storable*.

First, we look at the interface.

```
/* Storable.java
   Terry Marris  31 May 2001
*/

import java.util.*;

public interface Storable {
  public void write(Set s);
}
```

The line

```
public interface Storable {
```

specifies an interface named *Storable*. It looks just like a class header, but the keyword *class* is replaced with the keyword *interface*. And just like a class, braces { and } mark the beginning and end of the class body.

Between the class braces we have the method signature

```
public void write(Set s);
```

And that is it. No method implementation (notice the semi colon) and no instance variables.

The entire interface is saved in a file with the *.java* extension as usual, and can be compiled just like a regular java class. (Of course, it cannot be run because it has no *main()* method and no method implementations.)

Now we look at the *Copy* class. The class header includes the phrase *implements Storable*; now the class MUST implement *public void write(Set s);* We implement *title* as a string just to keep things simple.

```
/* Copy.java
   Terry Marris  31 May 2001
*/

import java.util.*;


public class Copy implements Storable {
  private static int nextNumber = 1;
  private int number;
  private String title;


  public Copy(String aTitle)
  {
    number = nextNumber;
    nextNumber++;
    title = aTitle;
  }


  public void write(Set s)
  {
    s.add(this);    // add yourself to the given set
  }


  public String toString()
  {
    return "Number: " + number + ", Title: " + title;
  }
```

The *main()* method creates a set and three *Copy* instances. It adds each *Copy* instance to the set. And then it prints out the contents of the set.

```
  public static void main(String[] s)
  {
    Set copySet = new HashSet();         // create the set

    Copy copy = new Copy("Z");
    copy.write(copySet);                 // message to copy:
                                         // write yourself on
                                         // the given set
    copy = new Copy("Java");
    copy.write(copySet);

    copy = new Copy("UML");
    copy.write(copySet);

    Iterator it = copySet.iterator();    // view the contents
    while (it.hasNext()) {               // of the set
      Object obj = it.next();
      copy = (Copy)obj;
      System.out.println(copy);
    }
  }
}    // end of class
```

Output:

```
Number: 2, Title: Java
Number: 1, Title: Z
Number: 3, Title: UML
```

16.7 PROPERTIES OF INTERFACES

We illustrate some of the properties of an interface. The *Account* interface has two implementations, *Credit* and *Overdrawn*.

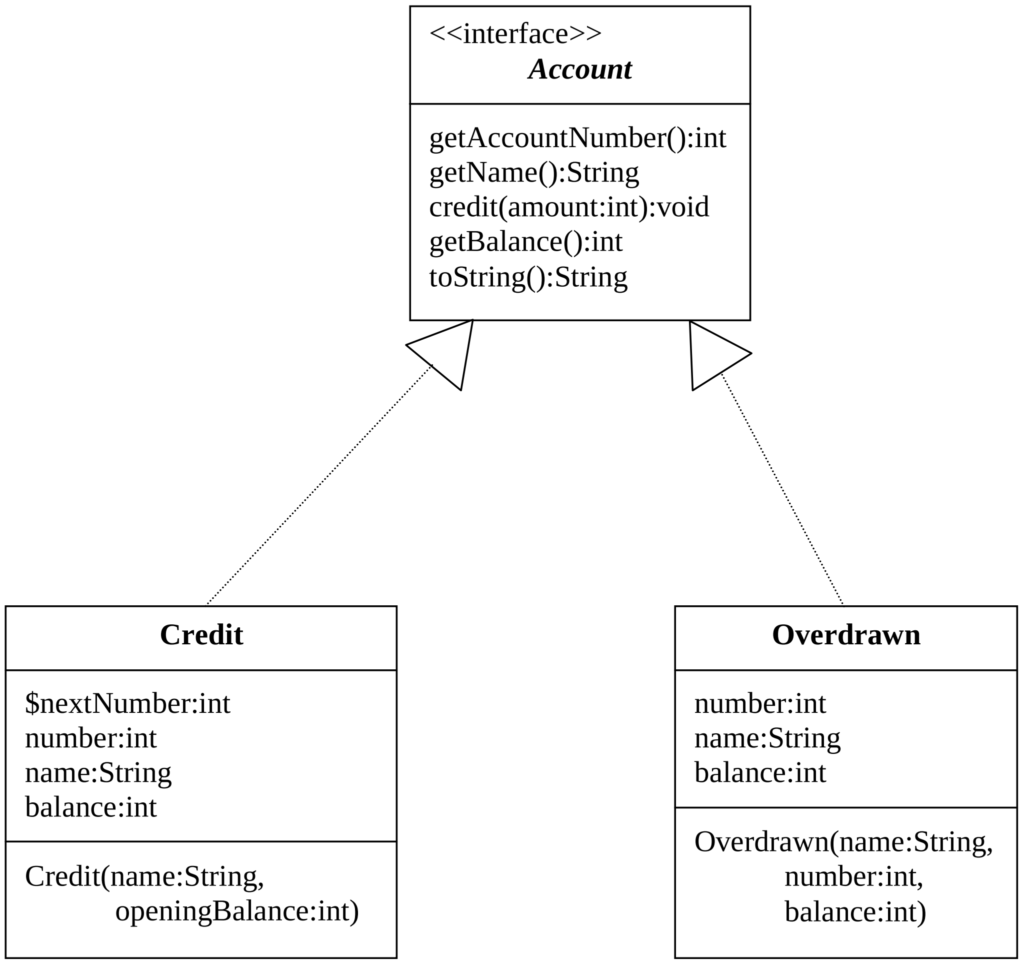

Figure 16.2 *Credit and Overdrawn both implement Account interface*

The operations defined in *Account* are not repeated in *Credit* or in *Overdrawn* because both these classes are obliged to provide an implementation of all these operations.

Each class can implement an operation in its own way; but the method signature must remain unchanged. Of course, each class may have additional methods. And each class must define its own instance variables.

We look first at the interface. It comprises just a list of method signatures.

```
/* Account.java
   Terry Marris  31 May 2001
*/


public interface Account {
  public int getAccountNumber();
  public String getName();
  public void credit(int amount);
  public int getBalance();
  public String toString();
}
```

Notice the semi-colons that follow each method signature; these are here because the methods are not implemented.

Now we look at the *Credit* class. It automatically generates a unique account number. It implements the usual setting and getting methods. The *credit(int)* method increases (or decreases) the amount stored in *balance* by messages such as

```
Account anAccount = new Credit("tom", 10);
...
anAccount.credit(15);    // adds 15 to balance
anAccount.credit(-2);    // subtracts 2 from balance
```

```
/* Credit.java
   Terry Marris  31 May 2001
*/


public class Credit implements Account {
  private static int nextNumber = 1001;
  private int number;
  private String name;
  private int balance;


  public Credit(String aName, int openingBalance)
  {
    number = nextNumber;
    name = aName;
    balance = openingBalance;
  }


  public int getAccountNumber()
  {
    return number;
  }


  public String getName()
  {
    return name;
  }


  public void credit(int amount)
  {
    balance += amount;
  }


  public int getBalance()
  {
    return balance;
  }


  public String toString()
  {
    return "Name: " + name + ", Number: " + number +
           ", Balance: " + balance;
  }
}
```

Now we look at the *Overdrawn* class. Essentially, we want a *Credit* instance to become an *Overdrawn* instance whenever the balance becomes less than zero. So *Overdrawn* does not generate its own account number; it relies on one being supplied to it as an argument value to the constructor. All the methods are implemented in the same way as they are in the *Credit* class, except the *toString()* method: it includes an ***Overdrawn*** caption.

```
/* Overdrawn.java
   Terry Marris 31 May 2001
*/

public class Overdrawn implements Account {
  private int number;
  private String name;
  private int balance;

  public Overdrawn(String aName, int aNumber, int aBalance)
  {
    number = aNumber;
    name = aName;
    balance = aBalance;
  }

  public int getAccountNumber()
  {
    return number;
  }

  public String getName()
  {
    return name;
  }

  public void credit(int amount)
  {
    balance += amount;
  }

  public int getBalance()
  {
    return balance;
  }

  public String toString()
  {
    return "**Overdrawn**  " + "Name: " + name +
           ", Number: " + number + ", Balance: " + balance;
  }
```

The *main()* method creates *Credit* account instance with opening balance 10 and displays it.

```
Account tomsAccount = new Credit("tom", 10);
System.out.println(tomsAccount);
```

We specify that *tomsAccount* is an object variable of type *Account* with
Account tomsAccount ... We create a *Credit* instance with a call to *new*.

After displaying the state of *tomsAccount*, *main()* goes on to decrease the balance in the account by 11, so that the account now becomes overdrawn.

```
tomsAccount.credit(-11);
```

When the account becomes overdrawn we create an *Overdrawn* instance with

```
Account overdrawn = new Overdrawn(
                         tomsAccount.getName(),
                         tomsAccount.getAccountNumber(),
                         tomsAccount.getBalance());
```

Then we assign the *Overdrawn* instance to the original account.

```
tomsAccount = overdrawn;
```

This is possible because *tomsAccount* and *overdrawn* are both *Account* object variables.

Here is *main()* in its entirety and its output.

```
  public static void main(String[] s)
  {
    Account tomsAccount = new Credit("tom", 10);
    System.out.println(tomsAccount);

    tomsAccount.credit(-11);
    if (tomsAccount.getBalance() < 0) {
      Account overdrawn = new Overdrawn(
                              tomsAccount.getName(),
                              tomsAccount.getAccountNumber(),
                              tomsAccount.getBalance());
      tomsAccount = overdrawn;
      System.out.println(tomsAccount);
    }
  }
}    // end of class
```

Output:

```
Name: tom, Number: 1001, Balance: 10
**Overdrawn**  Name: tom, Number: 1001, Balance: -1
```

An interface can extend another interface, so you can have chains of interfaces. Classes can also implement multiple interfaces. And a class may both extend another class and implement several interfaces

class AClass implements ASuperClass implements Interface1, Inteface2 {
...

We shall see cases of these constructs in parts Two and Three (Data Structures and Graphical User Interfaces) of the Java notes.

16.8 FURTHER READING

FOWLER & SCOTT *UML Distilled* pp 85
HORSTMANN & CORNELL Core Java 2 Volume 1 pp 225
www.java.sun.com/docs/books/tutorial

In the next chapter we look at the *HashMap* class, which implements the *Map* interface.

16.9 REVIEW

An interface contains a set of method signatures. A class that implements an interface must implement all the methods declared in that interface. An interface name can be used anywhere a type can be used.

16.10 EXERCISES

1 What is wrong with the following interface?

```
public interface X {
  public void m()
  {
    System.out.println("Hi Mom");
  }
}
```

2 Fix the interface shown in question 1 above.

3 Is the following interface valid?

```
public interface Marker {
}
```

Hint: try it out.