

CS1Bh Lecture Note 11

Interfaces and Abstract Classes

We have already seen some examples of *interfaces* in Java. An interface describes some behaviour which is to be implemented by a class. In this lecture we will examine interfaces in some more detail, and introduce *abstract classes*. An abstract class lies somewhere between an interface and an ordinary class: some of its methods may omit definitions (method bodies), being declared **abstract**.

11.1 Interfaces and implementations

An interface defines some behaviour which can be *implemented* by a class. We've already seen examples of this. An example from the Java API is the `Comparable` interface which describes the `compareTo` method. Many classes implement this interface; the meaning of the `compareTo` method is (informally) fixed by the interface.

Here's another simple interface, which defines the ability to send an email message:

```
1  interface Emailable {
2      void sendEmail(String msg); // Send me an email message
3  }
```

The `Emailable` interface specifies just one method, `sendEmail`, which sends the message `msg` to somewhere appropriate for the object. Supposing that University-wide student email addresses are based on matriculation numbers, we can give an implementation of `Emailable` for our familiar `Student` class:

```
1  class Student implements Emailable {
2      String forename;
3      String surname;
4      String matricno;
5      // Send an email message to SMS account of student
6      public void sendEmail(String msg) {
7          System.out.println("To: " + matricno +
8                              "@sms.ed.ac.uk" + "\n" + msg);
9      }
10 }
```

(In fact, this implementation just prints out the email message to the output stream, but this is safer for testing purposes than really sending emails!)

A Java interface defines a set of methods but does not implement them. A method appears as a header only, giving its return type, name, and argument list. No method bodies are allowed in an interface. Notice that the method `sendEmail` was declared **public** in the implementation but not the interface. All methods in interfaces must be public — a private interface method would be rather pointless, since an interface specifies a way of using some behaviour of the class. So Java allows you to omit the **public** modifier in the interface for brevity.

You can imagine an interface as a sort of contract to provide some behaviour, and a *protocol* for implementing that behaviour in terms of method calls. We use interfaces whenever we want to encapsulate some useful common behaviour across diverse classes; notice that the implementation relation defined by **implements** lies apart from the class hierarchy defined by **extends**. For example, apart from sending email messages to people, we can send special control messages to devices:

```
1  class Printer implements Emailable {
2      String name;
3      String location;
4      // Send a control message to the printer
5      public void sendEmail(String msg) {
6          System.out.println("To: printer-" + name + "\n" + msg);
7      }
8  }
```

A `Printer` object consists of a name and a location. A systems administrator might choose to provide an email interface to printers, using a special local email account for the printer based on its name. Then we could send email messages containing special configuration or printing instructions to the printer, using the `sendEmail` method.

This example is rather artificial, but it illustrates that although a printer is something quite different from a person (so `Printer` appears far away from `Student` in the class hierarchy), they can still have some behaviour in common. Interfaces are used to capture capabilities of a class: a “can-do” relationship, compared with the closer “is-an” subclassing relationship. Because the interface relationship is looser, a class may implement any number of interfaces; they are separated by commas in the class header. By contrast, a class extends exactly one direct superclass; if **extends** is not used, the superclass is `Object`. See the exercise at the end of the note to try implementing more interfaces in the `Student` class.

11.2 Constants in interfaces

As well as methods, an interface may contain constants. Recall that public constants in Java are declared with the modifiers **public static final**. Since interfaces are not allowed to have any other kind of variables, the modifiers are again optional. Here is an interface which describes a more general behaviour than sending emails:

```

1  interface Messagable {
2      int URGENT_PRIORITY = 0;
3      int MEDIUM_PRIORITY = 1;
4      int NONURGENT_PRIORITY = 2;
5
6      // Send a message with a given priority
7      void sendMessage(int priority, String msg);
8  }

```

The constants given in an interface are available for use by any class which implements the interface. Here's one:

```

1  class StudentWithMobile extends Student implements Messagable {
2      MobilePhone mp;
3      public void sendMessage(int priority, String msg) {
4          if (priority == URGENT_PRIORITY) {
5              mp.sendMessage(msg);
6          } else {
7              sendEmail(msg);
8          }
9      }
10 }

```

We assume that the `MobilePhone` object `mp` provides a method `sendMessage`. Urgent messages may be better sent as text messages instead of emails.

11.3 Using and extending interfaces

Just as we can use classes as types for variables and method arguments or results, so we can use interfaces as types. In fact, to compile `StudentWithMobile`, I defined `MobilePhone` as an interface containing the header `void sendMessage(String msg);`. So the interface `MobilePhone` was used as a type for the `mp` field.

Using interfaces as types allows additional flexibility for polymorphism. When a class `C` is used as a type for a parameter in some method header, the supplied argument can be an object from *any subclass* of `C`. The compiler selects the most appropriate method, automatically upcasting the object if necessary. When an interface `I` is used as a type, the method argument can be an object from *any implementing class* for `I`.

Interfaces stand apart from the usual class hierarchy, but in fact have their own hierarchy. We can declare an interface `I` that **extends** a previously define interfaces, or several previously defined interfaces (in contrast with **extends** for classes):

```

interface Controllable extends Messagable, Queryable { ... }

```

Then any class which **implements** `Controllable` must also implement the interfaces `Messagable` and `Queryable`.

11.4 Abstract classes

An *abstract class* lies somewhere between an interface and an ordinary class. Some methods may be declared with the **abstract** modifier, which means that no definition needs to be given. (In an interface, no method definitions are given, so the **abstract** modifier can be omitted). Usually an abstract class has a mixture of abstract and concrete methods:

```
1  abstract class ConnectedPerson implements Messagable {
2      public void sendMessage(int priority, String msg) {
3          switch (priority) {
4              case URGENT_PRIORITY: sendTextMessage(msg); break;
5              case MEDIUM_PRIORITY: sendEmail(msg); break;
6              default:
7                  sendFax(msg);
8          }
9      abstract public void sendFax(String msg);
10     abstract public void sendEmail(String msg);
11     abstract public void sendTextMessage(String msg);
12 }
```

A class which implements `ConnectedPerson` must provide implementations of the three delivery mechanisms. These behaviours are used in the standard `sendMessage`.

An abstract class appears in the main class hierarchy — it can extend another class, concrete or abstract — but as with interfaces, it is not possible to instantiate an abstract class. Writing `new ConnectedPerson()` will give a compiler error message. Like interfaces, abstract classes can also be used as object types.

11.5 Software engineering issues

Interfaces and abstract classes can be useful when building Java programs. We can use an interface instead of a class when some code has not yet been written, as with `MobilePhone`. This allows a programming task to be broken down and split amongst several people: once the interfaces are agreed, each engineer can work on implementing some pieces in isolation, and later combine their implementations.

Interfaces and abstract classes also help to foster *reuse*. A Java class library may contain many interfaces describing particular behaviours. A user of the library then only needs to understand a few interfaces to use many classes. A library developer will try to fit new classes into one or more of these standardized interfaces to help this.

A typical use of abstract classes is when there is a range of possible algorithms or data structures to choose from for some task, and some standard operations which are built on top. The higher level operations are given a fixed implementation, but the low-level algorithms may be implemented in several different ways and are declared **abstract**. Coding a new algorithm or data structure means only implementing the low-level operations in a new subclass, and not re-implementing the whole suite.

David Aspinall.

Javier Esparza, 2002/02/17 16:08:07.