CHAPTER

10 Improving Structure with Inheritance

Main concepts discussed in this chapter:

- inheritance
- subtyping
- substitution
- polymorphic variables

Java constructs discussed in this chapter:

`extends`, **`super`** (in constructor), cast, **`Object`**

In this chapter, we introduce some additional object-oriented constructs to improve the general structure of our applications. The main concepts we shall use to design better program structures are *inheritance* and *polymorphism*.

Both of these concepts are central to the idea of object orientation, and you will discover later how they appear in various forms in everything we discuss from now on. However, it is not only the following chapters that rely heavily on these concepts. Many of the constructs and techniques discussed in earlier chapters are influenced by aspects of inheritance and polymorphism, and we shall revisit some issues introduced earlier and gain a fuller understanding of the interconnections between different parts of the Java language.

Inheritance is a powerful construct that can be used to create solutions to a variety of different problems. As always, we will discuss the important aspects using an example. In this example, we will first introduce only some of the problems that are addressed by using inheritance structures, and discuss further uses and advantages of inheritance and polymorphism as we progress through this chapter.

The example we discuss to introduce these new structures is called *network*.

10.1 The *network* example

The *network* project implements a prototype of a small part of a social-network application. The part we are concentrating on is the *news feed*—the list of messages that should appear on screen when a user opens the network's main page.

Here, we will start small and simple, with a view to extending and growing the application later. Initially, we have only two types of posts appearing in our news feed: text posts (which we call *messages*), and photo posts consisting of a photo and a caption.

The part of the application that we are prototyping here is the engine that stores and displays these posts. The functionality that we want to provide with this prototype should include at least the following:

- It should allow us to create text and photo posts.
- Text posts consist of a message of arbitrary length, possibly spanning multiple lines. Photo posts consist of an image and a caption. Some additional details are stored with each post.
- It should store this information permanently so that it can be used later.
- It should provide a search function that allows us to find, for example, all posts by a certain user, or all photos within a given date range.
- It should allow us to display lists of posts, such as a list of the most recent posts, or a list of all posts by a given user.
- It should allow us to remove information.

The details we want to store for each message post are:

- the username of the author
- the text of the message
- a time stamp (time of posting)
- how many people like this post
- a list of comments on this post by other users

The details we want to store for each photo post are:

- the username of the author
- the filename of the image to display
- the caption for the photo (one line of text)
- a time stamp (time of posting)
- how many people like this post
- a list of comments on this post by other users

10.1.1 The *network* project: classes and objects

To implement the application, we first have to decide what classes to use to model this problem. In this case, some of the classes are easy to identify. It is quite straightforward to decide that we should have a class **`MessagePost`** to represent message posts, and a class **`PhotoPost`** to represent photo posts.

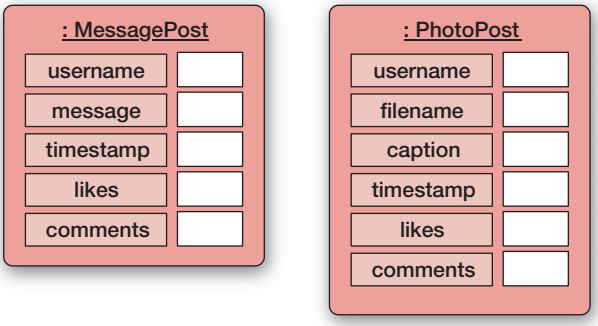

Figure 10.1
Fields in **`MessagePost`** and **`PhotoPost`** objects

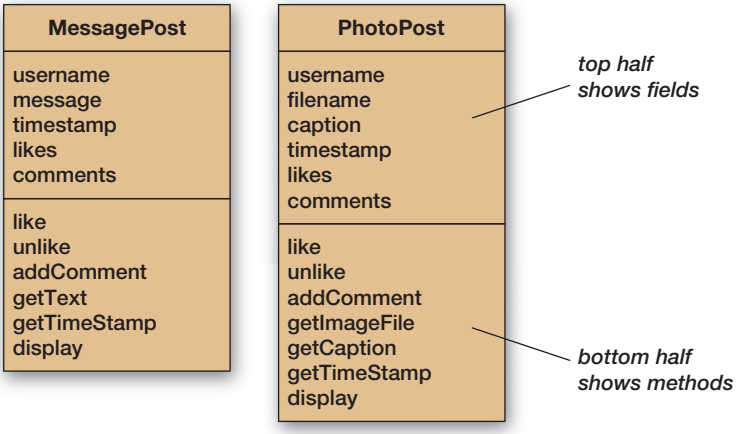

Figure 10.2
Details of the **`MessagePost`** and **`PhotoPost`** classes

Objects of these classes should then encapsulate all the data we want to store about these objects (Figure 10.1).

Some of these data items should probably also have accessor and mutator methods (Figure 10.2).[1] For our purpose, it is not important to decide on the exact details of all the methods right now, but just to get a first impression of the design of this application. In this figure, we have defined accessor and mutator methods for those fields that may change over time ("liking" or "unliking" a post and adding a comment) and assume for now that the other fields are set in the constructor. We have also added a method called **`display`** that will show details of a **`MessagePost`** or **`PhotoPost`** object.

[1] The notation style for class diagrams that is used in this book and in BlueJ is a subset of a widely used notation called UML. Although we do not use everything from UML (by far), we attempt to use UML notation for those things that we show. The UML style defines how fields and methods are shown in a class diagram. The class is divided into three parts that show (in this order from the top) the class name, the fields, and the methods.

Once we have defined the **MessagePost** and **PhotoPost** classes, we can create as many post objects as we need—one object per message post or photo post that we want to store. Apart from this, we then need another object: an object representing the complete news feed that can hold a collection of message posts and a collection of photo posts. For this, we shall create a class called **NewsFeed**.

The **NewsFeed** object could itself hold two collection objects (for example, of types **ArrayList<MessagePost>** and **ArrayList<PhotoPost>**). One of these collections can then hold all message posts, the other all photo posts. An object diagram for this model is shown in Figure 10.3.

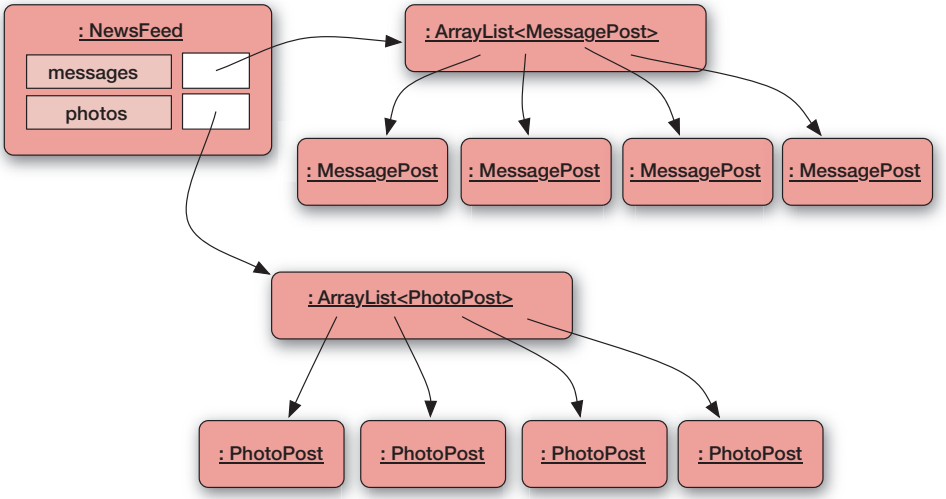

Figure 10.3
Objects in the *network* application

The corresponding class diagram, as BlueJ displays it, is shown in Figure 10.4. Note that BlueJ shows a slightly simplified diagram: classes from the standard Java library (**ArrayList** in this case) are not shown. Instead, the diagram focuses on user-defined classes. Also, BlueJ does not show field and method names in the diagram.

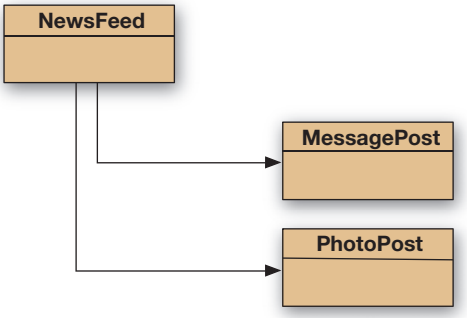

Figure 10.4
BlueJ class diagram of *network*

In practice, to implement the full *network* application, we would have more classes to handle things such as saving the data to a database and providing a user interface, most likely through a web browser. These are not very relevant to the present discussion, so we shall skip describing those for now, and concentrate on a more detailed discussion of the core classes mentioned here.

10.1.2 *Network* source code

So far, the design of the three current classes (**`MessagePost`**, **`PhotoPost`**, and **`NewsFeed`**) has been very straightforward. Translating these ideas into Java code is equally easy. Code 10.1 shows the source code of the **`MessagePost`** class. It defines the appropriate fields, sets in its constructor all the data items that are not expected to change over time, and provides accessor and mutator methods where appropriate. It also implements a first, simple version of the **`display`** method to show the post in the text terminal.

Code 10.1
Source code of the **`MessagePost`** class

```
import java.util.ArrayList;

/**
 * This class stores information about a post in a social network.
 * The main part of the post consists of a (possibly multi-line)
 * text message. Other data, such as author and time, are also stored.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.1
 */
public class MessagePost
{
    private String username;  // username of the post's author
    private String message;   // an arbitrarily long, multi-line message
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Constructor for objects of class MessagePost.
     *
     * @param author    The username of the author of this post.
     * @param text      The text of this post.
     */
    public MessagePost(String author, String text)
    {
        username = author;
        message = text;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }

    /**
     * Record one more "Like" indication from a user.
     */
    public void like()
    {
        likes++;
    }
```

Code 10.1 continued
Source code of the **MessagePost** class

```
/**
 * Record that a user has withdrawn his/her "Like" vote.
 */
public void unlike()
{
    if (likes > 0) {
        likes--;
    }
}

/**
 * Add a comment to this post.
 *
 * @param text  The new comment to add.
 */
public void addComment(String text)
{
    comments.add(text);
}

/**
 * Return the text of this post.
 *
 * @return The post's text.
 */
public String getText()
{
    return message;
}

/**
 * Return the time of creation of this post.
 *
 * @return The post's creation time, as a system time value.
 */
public long getTimeStamp()
{
    return timestamp;
}

/**
 * Display the details of this post.
 *
 * (Currently: Print to the text terminal. This is simulating display
 * in a web browser for now.)
 */
public void display()
{
    System.out.println(username);
    System.out.println(message);
    System.out.print(timeString(timestamp));

    if(likes > 0) {
        System.out.println("  -  " + likes + " people like this.");
    }
    else {
        System.out.println();
    }
```

Code 10.1 continued
Source code of the **MessagePost** class

```
        if(comments.isEmpty()) {
            System.out.println("    No comments.");
        }
        else {
            System.out.println("    " + comments.size() +
                               " comment(s). Click here to view.");
        }
    }

    /**
     * Create a string describing a time point in the past in terms
     * relative to current time, such as "30 seconds ago" or "7 minutes ago".
     * Currently, only seconds and minutes are used for the string.
     *
     * @param time  The time value to convert (in system milliseconds)
     * @return      A relative time string for the given time
     */
    private String timeString(long time)
    {
        long current = System.currentTimeMillis();
        long pastMillis = current - time;      // time passed in milliseconds
        long seconds = pastMillis/1000;
        long minutes = seconds/60;
        if(minutes > 0) {
            return minutes + " minutes ago";
        }
        else {
            return seconds + " seconds ago";
        }
    }
}
```

Some details are worth mentioning:

- Some simplifications have been made. For example, comments for a post are stored as strings. In a more complete version, we would probably use a custom class for comments, as comments also have additional detail such as an author and a time. The "like" count is stored as a simple integer. We are currently not recording which user liked a post. While these simplifications make our prototype incomplete, they are not relevant for our main discussion here, and we shall leave them as they are for now.
- The time stamp is stored as a single number, of type **`long`**. This reflects common practice. We can easily get the system time from the Java system, as a **`long`** value in milliseconds. We have also written a short method, called **`timeString`**, to convert this number into a relative time string, such as "5 minutes ago." In our final application, the system would have to use real time rather than system time, but again, system time is good enough for our prototype for now.

Note that we do not intend right now to make the implementation complete in any sense. It serves to provide a feel for what a class such as this might look like. We will use this as the basis for our following discussion of inheritance.

Now let us compare the **MessagePost** source code with the source code of class **PhotoPost**, shown in Code 10.2. Looking at both classes, we quickly notice that they are very similar. This is not surprising, because their purpose is similar: both are used to store information about news-feed posts, and the different types of post have a lot in common. They differ only in their details, such as some of their fields and corresponding accessors and the bodies of the **display** method.

Code 10.2
Source code of the **PhotoPost** class

```
import java.util.ArrayList;

/**
 * This class stores information about a post in a social network.
 * The main part of the post consists of a photo and a caption.
 * Other data, such as author and time, are also stored.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.1
 */
public class PhotoPost
{
    private String username;  // username of the post's author
    private String filename;  // the name of the image file
    private String caption;   // a one line image caption
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

    /**
     * Constructor for objects of class PhotoPost.
     *
     * @param author    The username of the author of this post.
     * @param filename  The filename of the image in this post.
     * @param caption   A caption for the image.
     */
    public PhotoPost(String author, String filename, String caption)
    {
        username = author;
        this.filename = filename;
        this.caption = caption;
        timestamp = System.currentTimeMillis();
        likes = 0;
        comments = new ArrayList<>();
    }

    /**
     * Record one more "Like" indication from a user.
     */
    public void like()
    {
        likes++;
    }

    /**
     * Record that a user has withdrawn his/her "Like" vote.
     */
```

Code 10.2 continued
Source code of the **PhotoPost** class

```
public void unlike()
{
    if (likes > 0) {
        likes--;
    }
}

/**
 * Add a comment to this post.
 *
 * @param text  The new comment to add.
 */
public void addComment(String text)
{
    comments.add(text);
}

/**
 * Return the file name of the image in this post.
 *
 * @return The post's image file name.
 */
public String getImageFile()
{
    return filename;
}

/**
 * Return the caption of the image of this post.
 *
 * @return The image's caption.
 */
public String getCaption()
{
    return caption;
}

/**
 * Return the time of creation of this post.
 *
 * @return The post's creation time, as a system time value.
 */
public long getTimeStamp()
{
    return timestamp;
}

/**
 * Display the details of this post.
 *
 * (Currently: Print to the text terminal. This is simulating display
 * in a web browser for now.)
 */
```

Code 10.2 continued
Source code of the **PhotoPost** class

```
    public void display()
    {
        System.out.println(username);
        System.out.println("  [" + filename + "]");
        System.out.println("  " + caption);
        System.out.print(timeString(timestamp));

        if(likes > 0) {
            System.out.println("  -  " + likes + " people like this.");
        }
        else {
            System.out.println();
        }

        if(comments.isEmpty()) {
            System.out.println("   No comments.");
        }
        else {
            System.out.println("   " + comments.size() +
                               " comment(s). Click here to view.");
        }
    }

    /**
     * Create a string describing a time point in the past in terms
     * relative to current time, such as "30 seconds ago" or "7 minutes ago".
     * Currently, only seconds and minutes are used for the string.
     *
     * @param time  The time value to convert (in system milliseconds)
     * @return      A relative time string for the given time
     */
    private String timeString(long time)
    {
        long current = System.currentTimeMillis();
        long pastMillis = current - time;       // time passed in milliseconds
        long seconds = pastMillis/1000;
        long minutes = seconds/60;
        if(minutes > 0) {
            return minutes + " minutes ago";
        }
        else {
            return seconds + " seconds ago";
        }
    }
}
```

Next, let us examine the source code of the **NewsFeed** class (Code 10.3). It, too, is quite simple. It defines two lists (each based on class **ArrayList**) to hold the collection of message posts and the collection of photo posts. The empty lists are created in the constructor. It then provides two methods for adding items: one for adding message posts, one for adding photo posts. The last method, named **show**, prints a list of all message and photo posts to the text terminal.

Code 10.3
Source code of the **NewsFeed** class

```
import java.util.ArrayList;

/**
 * The NewsFeed class stores news posts for the news feed in a social network
 * application.
 *
 * Display of the posts is currently simulated by printing the details to the
 * terminal. (Later, this should display in a browser.)
 *
 * This version does not save the data to disk, and it does not provide any
 * search or ordering functions.
 *
 * @author Michael Kölling and David J. Barnes
 * @version 0.1
 */
public class NewsFeed
{
    private ArrayList<MessagePost> messages;
    private ArrayList<PhotoPost> photos;

    /**
     * Construct an empty news feed.
     */
    public NewsFeed()
    {
        messages = new ArrayList<>();
        photos = new ArrayList<>();
    }

    /**
     * Add a text post to the news feed.
     *
     * @param text  The text post to be added.
     */
    public void addMessagePost(MessagePost message)
    {
        messages.add(message);
    }

    /**
     * Add a photo post to the news feed.
     *
     * @param photo  The photo post to be added.
     */
    public void addPhotoPost(PhotoPost photo)
    {
        photos.add(photo);
    }

    /**
     * Show the news feed. Currently: print the news feed details to the
     * terminal. (To do: replace this later with display in web browser.)
```

Code 10.3 continued
Source code of the **NewsFeed** class

```
    public void show()
    {
        // display all text posts
        for(MessagePost message : messages) {
            message.display();
            System.out.println();   // empty line between posts
        }

        // display all photos
        for(PhotoPost photo : photos) {
            photo.display();
            System.out.println();   // empty line between posts
        }
    }
}
```

This is by no means a complete application. It has no user interface yet (so it will not be usable outside BlueJ), and the data entered is not stored to the file system or in a database. This means that all data entered will be lost each time the application ends. There are no functions to sort the displayed list of posts—for example, by date and time or by relevance. Currently, we will always get messages first, in the order in which they were entered, followed by the photos. Also, the functions for entering and editing data, as well as searching for data and displaying it, are not flexible enough for what we would want from a real program.

However, this does not matter in our context. We can work on improving the application later. The basic structure is there, and it works. This is enough for us to discuss design problems and possible improvements.

Exercise 10.1 Open the project *network-v1*. It contains the classes exactly as we have discussed them here. Create some **`MessagePost`** objects and some **`PhotoPost`** objects. Create a **`NewsFeed`** object. Enter the posts into the news feed, and then display the feed's contents.

Exercise 10.2 Try the following. Create a **`MessagePost`** object. Enter it into the news feed. Display the news feed. You will see that the post has no associated comments. Add a comment to the **`MessagePost`** object on the object bench (the one you entered into the news feed). When you now list the news feed again, will the post listed there have a comment attached? Try it. Explain the behavior you observe.

10.1.3 Discussion of the *network* application

Even though our application is not yet complete, we have done the most important part. We have defined the core of the application—the data structure that stores the essential information.

This was fairly straightforward so far, and we could now go ahead and design the rest that is still missing. Before doing that, though, we will discuss the quality of the solution so far.