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.
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 the 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.
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.

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.
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.

```java
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

```java
/**
 * 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.println(timeStamp(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";
    }
}

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.

```java
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.
     */
```
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.)
 */
public void display()
{
    System.out.println(username);
    System.out.println(" [" + filename + "]");
    System.out.println(" " + caption);
    System.out.println(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

```java
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.)
     */
```
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.
There are several fundamental problems with our current solution. The most obvious one is code duplication.

We have noticed above that the MessagePost and PhotoPost classes are very similar. In fact, the majority of the classes’ source code is identical, with only a few differences. We have already discussed the problems associated with code duplication in Chapter 8. Apart from the annoying fact that we have to write everything twice (or copy and paste, then go through and fix all the differences), there are often problems associated with maintaining duplicated code. Many possible changes would have to be done twice. If, for example, the type of the comment list is changed from ArrayList<String> to ArrayList<Comment> (so that more details can be stored), this change has to be made once in the MessagePost class and again in the PhotoPost class. In addition, associated with maintenance of code duplication is always the danger of introducing errors, because the maintenance programmer might not realize that an identical change is needed at a second (or third) location.

There is another spot where we have code duplication: in the NewsFeed class. We can see that everything in that class is done twice—once for message posts, and once for photo posts. The class defines two list variables, then creates two list objects, defines two add methods, and has two almost-identical blocks of code in the show method to print out the lists.

The problems with this duplication become clear when we analyze what we would have to do to add another type of post to this program. Imagine that we want to store not only text messages and photo posts, but also activity posts. Activity posts can be automatically generated and inform us about an activity of one of our contacts, such as “Fred has changed his profile picture” or “Jacob is now friends with Feena.” Activity posts seem similar enough that it should be easy to modify our application to do this. We would introduce another class, ActivityPost, and essentially write a third version of the source code that we already have in the MessagePost and PhotoPost classes. Then we have to work through the NewsFeed class and add another list variable, another list object, another add method, and another loop in the show method.

We would have to do the same for a fourth type of post. The more we do this, the more the code-duplication problem increases, and the harder it becomes to make changes later. When we feel uncomfortable about a situation such as this one, it is often a good indicator that there may be a better alternative approach. For this particular case, the solution is found in object-oriented languages. They provide a distinctive feature that has a big impact on programs involving sets of similar classes. In the following sections, we will introduce this feature, which is called inheritance.

**Using inheritance**

Inheritance is a mechanism that provides us with a solution to our problem of duplication. The idea is simple: instead of defining the MessagePost and PhotoPost classes completely independently, we first define a class that contains everything these two have in common. We shall call this class Post. Then we can declare that a MessagePost is a Post and a PhotoPost is a Post. Finally, we add those extra details needed for a message post to the MessagePost class, and those for a photo post to the PhotoPost class. The essential feature of this technique is that we need to describe the common features only once.
to access or change private fields in its superclass, then the superclass would need to provide appropriate accessor and/or mutator methods. However, an object of a subclass may call any public methods defined in its superclass as if they were defined locally in the subclass—no variable is needed, because the methods are all part of the same object.

This issue of access rights between super- and subclasses is one we will discuss further in Chapter 11, when we introduce the **protected** modifier.

**Exercise 10.4** Open the project `network-v2`. This project contains a version of the `network` application, rewritten to use inheritance, as described above. Note that the class diagram displays the inheritance relationship. Open the source code of the `MessagePost` class and remove the “extends `Post`” phrase. Close the editor. What changes do you observe in the class diagram? Add the “extends `Post`” phrase again.

**Exercise 10.5** Create a `MessagePost` object. Call some of its methods. Can you call the inherited methods (for example, `addComment`)? What do you observe about the inherited methods?

**Exercise 10.6** In order to illustrate that a subclass can access non-private elements of its superclass without any special syntax, try the following slightly artificial modification to the `MessagePost` and `Post` classes. Create a method called `printShortSummary` in the `MessagePost` class. Its task is to print just the phrase “Message post from `NAME`”, where `NAME` should show the name of the author. However, because the `username` field is private in the `Post` class, it will be necessary to add a public `getUserName` method to `Post`. Call this method from `printShortSummary` to access the name for printing. Remember that no special syntax is required when a subclass calls a superclass method. Try out your solution by creating a `MessagePost` object. Implement a similar method in the `PhotoPost` class.

### 10.4.2 Inheritance and initialization

When we create an object, the constructor of that object takes care of initializing all object fields to some reasonable state. We have to look more closely at how this is done in classes that inherit from other classes.

When we create a `MessagePost` object, we pass two parameters to the message post’s constructor: the name of the author and the message text. One of these contains a value for a field defined in class `Post`, and the other a value for a field defined in class `MessagePost`. All of these fields must be correctly initialized, and Code 10.4 shows the code segments that are used to achieve this in Java.

Several observations can be made here. First, the class `Post` has a constructor, even though we do not intend to create an instance of class `Post` directly.\(^2\) This constructor receives the

\(^2\) Currently, there is nothing actually preventing us from creating a `Post` object, although that was not our intention when we designed these classes. In Chapter 12, we shall see some techniques that allow us to make sure that `Post` objects cannot be created directly, but only `MessagePost` or `PhotoPost` objects.
Figure 10.5 shows a class diagram for this new structure. At the top, it shows the class `Post`, which defines all fields and methods that are common to all posts (messages and photos). Below the `Post` class, it shows the `MessagePost` and `PhotoPost` classes, which hold only those fields and methods that are unique to each particular class.

This new feature of object-oriented programming requires some new terminology. In a situation such as this one, we say that the class `MessagePost` *inherits from* class `Post`. Class `PhotoPost` also inherits from `Post`. In the vernacular of Java programs, the expression "class `MessagePost extends class Post`" could be used, because Java uses an `extends` keyword to define the inheritance relationship (as we shall see shortly). The arrows in the class diagram (usually drawn with hollow arrow heads) represent the inheritance relationship.

Class `Post` (the class that the others inherit from) is called the *parent class* or *superclass*. The inheriting classes (`MessagePost` and `PhotoPost` in this example) are referred to as *child classes* or *subclasses*. In this book, we will use the terms "superclass" and "subclass" to refer to the classes in an inheritance relationship.

Inheritance is sometimes also called an *is-a* relationship. The reason is that a subclass is a specialization of a superclass. We can say that "a message post *is a* post" and "a photo post *is a* post."

The purpose of using inheritance is now fairly obvious. Instances of class `MessagePost` will have all fields defined in `MessagePost` *and* in class `Post`. (`MessagePost` inherits the fields from `Post`.) Instances of `PhotoPost` will have all fields defined in `PhotoPost` and in `Post`. Thus, we achieve the same as before, but we need to define the fields `username, timestamp, likes,` and `comments` only once, while being able to use them in two different places.
The same holds true for methods: instances of subclasses have all methods defined in both the superclass and the subclass. In general, we can say: because a message post is a post, a message-post object has everything that a post has, and more. And because a photo post is also a post, it has everything that a post has, and more.

Thus, inheritance allows us to create two classes that are quite similar, while avoiding the need to write the identical part twice. Inheritance has a number of other advantages, which we discuss below. First, however, we will take another, more general look at inheritance hierarchies.

**Inheritance hierarchies**

Inheritance can be used much more generally than shown in the example above. More than two subclasses can inherit from the same superclass, and a subclass can, in turn, be a superclass to other subclasses. The classes then form an inheritance hierarchy.

The best-known example of an inheritance hierarchy is probably the classification of species used by biologists. A small part is shown in Figure 10.6. We can see that a poodle is a dog, which is a mammal, which is an animal.

We know some things about poodles—for example, that they are alive, they can bark, they eat meat, and they give birth to live young. On closer inspection, we see that we know some of these things not because they are poodles, but because they are dogs, mammals, or animals. An instance of class **Poodle** (an actual poodle) has all the characteristics of a poodle, a dog, a mammal, and an animal, because a poodle is a dog, which is a mammal, and so on.

The principle is simple: inheritance is an abstraction technique that lets us categorize classes of objects under certain criteria and helps us specify the characteristics of these classes.

**Exercise 10.3** Draw an inheritance hierarchy for the people in your place of study or work. For example, if you are a university student, then your university probably has students (first-year students, second-year students, . . .), professors, tutors, office personnel, etc.

![Figure 10.6](image-url)

An example of an inheritance hierarchy
10.4 Inheritance in Java

Before discussing more details of inheritance, we will have a look at how inheritance is expressed in the Java language. Here is a segment of the source code of the Post class:

```java
public class Post {
    private String username; // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;
    // Constructors and methods omitted.
}
```

There is nothing special about this class so far. It starts with a normal class definition and defines Post’s fields in the usual way. Next, we examine the source code of the MessagePost class:

```java
public class MessagePost extends Post {
    private String message;
    // Constructors and methods omitted.
}
```

There are two things worth noting here. First, the keyword `extends` defines the inheritance relationship. The phrase “extends Post” specifies that this class is a subclass of the Post class. Second, the MessagePost class defines only those fields that are unique to MessagePost objects (only `message` in this case). The fields from Post are inherited and do not need to be listed here. Objects of class MessagePost will nonetheless have fields for `username`, `timestamp`, and so on.

Next, let us have a look at the source code of class PhotoPost:

```java
public class PhotoPost extends Post {
    private String filename;
    private String caption;
    // Constructors and methods omitted.
}
```

This class follows the same pattern as the MessagePost class. It uses the `extends` keyword to define itself as a subclass of Post and defines its own additional fields.

10.4.1 Inheritance and access rights

To objects of other classes, MessagePost or PhotoPost objects appear just like all other types of objects. As a consequence, members defined as `public` in either the superclass or subclass portions will be accessible to objects of other classes, but members defined as `private` will be inaccessible.

In fact, the rule on privacy also applies between a subclass and its superclass: a subclass cannot access private members of its superclass. It follows that if a subclass method needed
public class Post {
    private String username; // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;

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

    Methods omitted.
}

public class MessagePost extends Post {
    private String message; // an arbitrarily long, multi-line message

    /**
     * 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) {
        super(author);
        message = text;
    }

    Methods omitted.
}

parameters needed to initialize the Post fields, and it contains the code to do this initialization. Second, the MessagePost constructor receives parameters needed to initialize both Post and MessagePost fields. It then contains the following line of code:

    super(author);
The keyword super is a call from the subclass constructor to the constructor of the superclass. Its effect is that the Post constructor is executed as part of the MessagePost constructor’s execution. When we create a message post, the MessagePost constructor is called, which, in turn, as its first statement, calls the Post constructor. The Post constructor initializes the post’s fields, and then returns to the MessagePost constructor, which initializes the remaining field defined in the MessagePost class. For this to work, those parameters needed for the initialization of the post fields are passed on to the superclass constructor as parameters to the super call.

In Java, a subclass constructor must always call the superclass constructor as its first statement. If you do not write a call to a superclass constructor, the Java compiler will insert a superclass call automatically, to ensure that the superclass fields are properly initialized. The inserted call is equivalent to writing

    super();

Inserting this call automatically works only if the superclass has a constructor without parameters (because the compiler cannot guess what parameter values should be passed). Otherwise, an error will be reported.

In general, it is a good idea to always include explicit superclass calls in your constructors, even if it is one that the compiler could generate automatically. We consider this good style, because it avoids the possibility of misinterpretation and confusion in case a reader is not aware of the automatic code generation.

**Exercise 10.7** Set a breakpoint in the first line of the MessagePost class’s constructor. Then create a MessagePost object. When the debugger window pops up, use Step Into to step through the code. Observe the instance fields and their initialization. Describe your observations.

### 10.5 Network: adding other post types

Now that we have our inheritance hierarchy set up for the network project so that the common elements of the items are in the Post class, it becomes a lot easier to add other types of posts. For instance, we might want to add event posts, which consist of a description of a standard event (e.g., “Fred has joined the ‘Neal Stephenson fans’ group.”). Standard events might be a user joining a group, a user becoming friends with another, or a user changing their profile picture. To achieve this, we can now define a new subclass of Post named EventPost (Figure 10.7). Because EventPost is a subclass of Post, it automatically inherits all fields and methods that we have already defined in Post. Thus, EventPost objects already have a username, a timestamp, a likes counter, and comments. We can then concentrate on adding attributes that are specific to event posts, such as the event type. The event type might be stored as an enumeration constant (see Chapter 8) or as a string describing the event.

This is an example of how inheritance enables us to reuse existing work. We can reuse the code that we have written for photo posts and message posts (in the Post class) so that it
Also works for the `EventPost` class. The ability to reuse existing software components is one of the great benefits that we get from the inheritance facility. We will discuss this in more detail later.

This reuse has the effect that a lot less new code is needed when we now introduce additional post types. Because new post types can be defined as subclasses of `Post`, only the code that is actually different from `Post` has to be added.

Now imagine that we change the requirements a bit: event posts in our network application will not have a “Like” button or comments attached. They are for information only. How do we achieve this? Currently, because `EventPost` is a subclass of `Post`, it automatically inherits the `likes` and `comments` fields. Is this a problem?

We could leave everything as it is and decide to never display the likes count or comments for event posts—just ignore the fields. This does not feel right. Having the fields present but unused invites problems. Someday, a maintenance programmer will come along who does not realize that these fields should not be used and try to process them.

Or we could write `EventPost` without inheriting from `Post`. But then we are back to code duplication for the `username` and `timestamp` fields and their methods.

The solution is to refactor the class hierarchy. We can introduce a new superclass for all posts that have comments attached (named `CommentedPost`), which is a subclass of `Post` (Figure 10.8). We then shift the `likes` and `comments` fields from the `Post` class to this new class. `MessagePost` and `PhotoPost` are now subclasses of our new `CommentedPost` class, while `EventPost` inherits from `Posts` directly. `MessagePost` objects inherit everything from both superclasses and have the same fields and methods as before. Objects of class `EventPost` will inherit the `username` and `timestamp`, but not the comments.

This is a very common situation in designing class hierarchies. When the hierarchy does not seem to fit properly, we have to refactor the hierarchy.
Classes that are not intended to be used to create instances, but whose purpose is exclusively to serve as superclasses for other classes (such as `Post` and `CommentedPost`), are called abstract classes. We shall investigate this in more detail in Chapter 12.

**Exercise 10.8** Open the `network-v2` project. Add a class for event posts to the project. Create some event-post objects and test that all methods work as expected.

### Advantages of inheritance (so far)

We have seen several advantages of using inheritance for the `network` application. Before we explore other aspects of inheritance, we shall summarize the general advantages we have encountered so far:

- **Avoiding code duplication** The use of inheritance avoids the need to write identical or very similar copies of code twice (or even more often).

- **Code reuse** Existing code can be reused. If a class similar to the one we need already exists, we can sometimes subclass the existing class and reuse some of the existing code, rather than having to implement everything again.
- **Easier maintenance** Maintaining the application becomes easier, because the relationship between the classes is clearly expressed. A change to a field or a method that is shared between different types of subclasses needs to be made only once.

- **Extendibility** Using inheritance, it becomes much easier to extend an existing application in certain ways.

**Exercise 10.9** Order these items into an inheritance hierarchy: apple, ice cream, bread, fruit, food item, cereal, orange, dessert, chocolate mousse, baguette.

**Exercise 10.10** In what inheritance relationship might a *touch pad* and a *mouse* be? (We are talking about computer input devices here, not a small furry mammal.)

**Exercise 10.11** Sometimes things are more difficult than they first seem. Consider this: In what kind of inheritance relationship are *Rectangle* and *Square*? What are the arguments? Discuss.

### 10.7 Subtyping

The one thing we have not yet investigated is how the code in the *NewsFeed* class was changed when we modified our project to use inheritance. Code 10.5 shows the full source code of class *NewsFeed*. We can compare this with the original source shown in Code 10.3.

```java
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 0.2
 */
public class NewsFeed
{
    private ArrayList<Post> posts;

    /**
     * Construct an empty news feed.
     */
```
Code 10.5
Source code of the NewsFeed class (second version)

```java
public NewsFeed()
{
    posts = new ArrayList<>();
}

/**
 * Add a post to the news feed.
 * @param post The post to be added.
 */
public void addPost(Post post)
{
    posts.add(post);
}

/**
 * Show the news feed. Currently: print the news feed details to the terminal. (To do: replace this later with display in web browser.)
 */
public void show()
{
    // display all posts
    for(Post post : posts) {
        post.display();
        System.out.println();  // empty line between posts
    }
}
```

As we can see, the code has become significantly shorter and simpler since our change to use inheritance. Where in the first version (Code 10.3) everything had to be done twice, it now exists only once. We have only one collection, only one method to add posts, and one loop in the `show` method.

The reason why we could shorten the source code is that, in the new version, we can use the type `Post` where we previously used `MessagePost` and `PhotoPost`. We investigate this first by examining the `addPost` method.

In our first version, we had two methods to add posts to the news feed. They had the following headers:

```java
public void addMessagePost(MessagePost message)
public void addPhotoPost(PhotoPost photo)
```

In our new version, we have a single method to serve the same purpose:

```java
public void addPost(Post post)
```

The parameters in the original version are defined with the types `MessagePost` and `PhotoPost`, ensuring that we pass `MessagePost` and `PhotoPost` objects to these methods, because actual parameter types must match the formal parameter types. So far, we have interpreted the requirement that parameter types must match as meaning “must be of
the same type”—for instance, that the type name of an actual parameter must be the same as the type name of the corresponding formal parameter. This is only part of the truth, in fact, because an object of a subclass can be used wherever its superclass type is required.

10.7.1 Subclasses and subtypes

We have discussed earlier that classes define types. The type of an object that was created from class `MessagePost` is `MessagePost`. We also just discussed that classes may have subclasses. Thus, the types defined by the classes can have subtypes. In our example, the type `MessagePost` is a subtype of type `Post`.

10.7.2 Subtyping and assignment

When we want to assign an object to a variable, the type of the object must match the type of the variable. For example,

```java
Car myCar = new Car();
```

is a valid assignment, because an object of type `Car` is assigned to a variable declared to hold objects of type `Car`. Now that we know about inheritance, we must state the typing rule more completely: a variable can hold objects of its declared type, or of any subtype of its declared type.

Imagine that we have a class `Vehicle` with two subclasses, `Car` and `Bicycle` (Figure 10.9). In this case, the typing rule allows all the following assignments:

```java
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

The type of a variable declares what it can store. Declaring a variable of type `Vehicle` states that this variable can hold vehicles. But because a car is a vehicle, it is perfectly legal to store a car in a variable that is intended for vehicles. (Think of the variable as a garage: if someone tells you that you may park a vehicle in a garage, you would think that parking either a car or a bicycle in the garage would be okay.)

This principle is known as substitution. In object-oriented languages, we can substitute a subclass object where a superclass object is expected, because the subclass object is a special case of the superclass. If, for example, someone asks us to give them a pen, we can

---

**Figure 10.9**

An inheritance hierarchy
fulfill the request perfectly well by giving them a fountain pen or a ballpoint pen. Both fountain pen and ballpoint pen are subclasses of pen, so supplying either where an object of class `Pen` was expected is fine.

However, doing it the other way is not allowed:

```java
Car c1 = new Vehicle(); // this is an error!
```

This statement attempts to store a `Vehicle` object in a `Car` variable. Java will not allow this, and an error will be reported if you try to compile this statement. The variable is declared to be able to store cars. A vehicle, on the other hand, may or may not be a car—we do not know. Thus, the statement may be wrong, and therefore not allowed.

Similarly:

```java
Car c2 = new Bicycle(); // this is an error!
```

This is also an illegal statement. A bicycle is not a car (or, more formally, the type `Bicycle` is not a subtype of `Car`), and thus the assignment is not allowed.

**Exercise 10.12** Assume that we have four classes: `Person`, `Teacher`, `Student`, and `PhDStudent`. `Teacher` and `Student` are both subclasses of `Person`. `PhDStudent` is a subclass of `Student`.

a. Which of the following assignments are legal, and why or why not?

```java
Person p1 = new Student();
Person p2 = new PhDStudent();
PhDStudent phd1 = new Student();
Teacher t1 = new Person();
Student s1 = new PhDStudent();
```

b. Suppose that we have the following legal declarations and assignments:

```java
Person p1 = new Person();
Person p2 = new Person();
PhDStudent phd1 = new PhDStudent();
Teacher t1 = new Teacher();
Student s1 = new Student();
```

Based on those just mentioned, which of the following assignments are legal, and why or why not?

```java
s1 = p1
s1 = p2
p1 = s1;
t1 = s1;
s1 = phd1;
phd1 = s1;
```
Exercise 10.13 Test your answers to the previous question by creating bare-bones versions of the classes mentioned in that exercise and trying it out in BlueJ.

10.7.3 Subtyping and parameter passing

Passing a parameter (that is, assigning an actual parameter to a formal parameter variable) behaves in exactly the same way as an assignment to a variable. This is why we can pass an object of type `MessagePost` to a method that has a parameter of type `Post`. We have the following definition of the `addPost` method in class `NewsFeed`:

```java
public void addPost(Post post)
{
    . . .
}
```

We can now use this method to add message posts and photo posts to the feed:

```java
NewsFeed feed = new NewsFeed();
MessagePost message = new MessagePost(. . .);
PhotoPost photo = new PhotoPost(. . .);
feed.addPost(message);
feed.addPost(photo);
```

Because of subtyping rules, we need only one method (with a parameter of type `Post`) to add both `MessagePost` and `PhotoPost` objects.

We will discuss subtyping in more detail in the next chapter.

10.7.4 Polymorphic variables

Variables holding object types in Java are `polymorphic` variables. The term “polymorphic” (literally, many shapes) refers to the fact that a variable can hold objects of different types (namely, the declared type or any subtype of the declared type). Polymorphism appears in object-oriented languages in several contexts—polymorphic variables are just the first example. We will discuss other incarnations of polymorphism in more detail in the next chapter.

For now, we just observe how the use of a polymorphic variable helps us simplify our `show` method. The body of this method is

```java
for(Post post : posts) {
    post.display();
    System.out.println(); // empty line between posts
}
```

Here, we iterate through the list of posts (held in an `ArrayList` in the `posts` variable). We get out each post and then invoke its `display` method. Note that the actual posts that we get out of the list are of type `MessagePost` or `PhotoPost`, not of type `Post`. We can, however, use a loop variable of type `Post`, because variables are polymorphic.
The `post` variable is able to hold `MessagePost` and `PhotoPost` objects, because these are subtypes of `Post`.

Thus, the use of inheritance in this example has removed the need for two separate loops in the `show` method. Inheritance avoids code duplication not only in the server classes, but also in clients of those classes.

**Note** When doing the exercises, you may have noticed that the `show` method has a problem: not all details are printed out. Solving this problem requires some more explanation. We will provide this in the next chapter.

**Exercise 10.14** What has to change in the `NewsFeed` class when another `Post` subclass (for example, a class `EventPost`) is added? Why?

### 10.7.5 Casting

Sometimes the rule that we cannot assign from a supertype to a subtype is more restrictive than necessary. If we know that the supertype variable holds a subtype object, the assignment could actually be allowed. For example:

```java
Vehicle v;
Car c = new Car();
v = c;  // correct
c = v;  // error
```

The above statements would not compile: we get a compiler error in the last line, because assigning a `Vehicle` variable to a `Car` variable (supertype to subtype) is not allowed. However, if we execute these statements in sequence, we know that we could actually allow this assignment. We can see that the variable `v` actually contains an object of type `Car`, so the assignment to `c` would be okay. The compiler is not that smart. It translates the code line by line, so it looks at the last line in isolation without knowing what is currently stored in variable `v`. This is called type loss. The type of the object in `v` is actually `Car`, but the compiler does not know this.

We can get around this problem by explicitly telling the type system that the variable `v` holds a `Car` object. We do this using a cast operator:

```java
c = (Car) v;  // okay
```

The cast operator consists of the name of a type (here, `Car`) written in parentheses in front of a variable or an expression. Doing this will cause the compiler to believe that the object is a `Car`, and it will not report an error. At runtime, however, the Java system will check that it really is a `Car`. If we were careful, and it is truly is a `Car`, everything is fine. If the object in `v` is of another type, the runtime system will indicate an error (called a `ClassCastException`), and the program will stop.\(^3\)

---

\(^3\) Exceptions are discussed in detail in Chapter 14.
Now consider this code fragment, in which `Bicycle` is also a subclass of `Vehicle`:

```java
Vehicle v;
Car c;
Bicycle b;
c = new Car();
v = c; // okay
b = (Bicycle) c; // compile time error!
b = (Bicycle) v; // runtime error!
```

The last two assignments will both fail. The attempt to assign `c` to `b` (even with the cast) will be a compile-time error. The compiler notices that `Car` and `Bicycle` do not form a subtype/supertype relationship, so `c` can never hold a `Bicycle` object—the assignment could never work.

The attempt to assign `v` to `b` (with the cast) will be accepted at compile time, but will fail at runtime. `Vehicle` is a superclass of `Bicycle`, and thus `v` can potentially hold a `Bicycle` object. At runtime, however, it turns out that the object in `v` is not a `Bicycle` but a `Car`, and the program will terminate prematurely.

Casting should be avoided wherever possible, because it can lead to runtime errors, and that is clearly something we do not want. The compiler cannot help us to ensure correctness in this case.

In practice, casting is very rarely needed in a well-structured, object-oriented program. In almost all cases, when you use a cast in your code, you could restructure your code to avoid this cast and end up with a better-designed program. This usually involves replacing the cast with a polymorphic method call (more about this will be covered in the next chapter).

### 10.8 The Object class

All classes have a superclass. So far, it has appeared as if most classes we have seen do not have a superclass. In fact, while we can declare an explicit superclass for a class, all classes that have no superclass declaration implicitly inherit from a class called `Object`.

`Object` is a class from the Java standard library that serves as a superclass for all objects. Writing a class declaration such as

```java
public class Person {
    ...
}
```

is equivalent to writing

```java
public class Person extends Object {
    ...
}
```
The Java compiler automatically inserts the `Object` superclass for all classes without an explicit `extends` declaration, so it is never necessary to do this for yourself. Every single class (with the sole exception of the `Object` class itself) inherits from `Object`, either directly or indirectly. Figure 10.10 shows some randomly chosen classes to illustrate this.

Having a common superclass for all objects serves two purposes: First, we can declare polymorphic variables of type `Object` to hold any object. Having variables that can hold any object type is not often useful, but there are some situations where this can help. Second, the `Object` class can define some methods that are then automatically available for every existing object. Of particular importance are the methods `toString`, `equals`, and `hashCode` which `Object` defines. This second point becomes interesting a bit later, and we shall discuss this in more detail in the next chapter.

## 10.9 The collection hierarchy

The Java library uses inheritance extensively in the definition of the collections classes. Class `ArrayList`, for example, inherits from a class called `AbstractList`, which, in turn, inherits from `AbstractCollection`. We shall not discuss this hierarchy here, because it is described in detail at various easily accessible places. One good description is at Oracle's web site at [http://download.oracle.com/javase/tutorial/collections/index.html](http://download.oracle.com/javase/tutorial/collections/index.html).

Note that some details of this hierarchy require an understanding of Java `interfaces`. We discuss those in Chapter 12.

**Exercise 10.15** Use the documentation of the Java standard class libraries to find out about the inheritance hierarchy of the collection classes. Draw a diagram showing the hierarchy.
Summary

This chapter has presented a first view of inheritance. All classes in Java are arranged in an inheritance hierarchy. Each class may have an explicitly declared superclass, or it inherits implicitly from the class `Object`.

Subclasses usually represent specializations of superclasses. Because of this, the inheritance relationship is also referred to as an is-a relationship (a car is-a vehicle).

Subclasses inherit all fields and methods of a superclass. Objects of subclasses have all fields and methods declared in their own classes, as well as those from all superclasses. Inheritance relationships can be used to avoid code duplication, to reuse existing code, and to make an application more maintainable and extendable.

Subclasses also form subtypes, which leads to polymorphic variables. Subtype objects may be substituted for supertype objects, and variables are allowed to hold objects that are instances of subtypes of their declared type.

Inheritance allows the design of class structures that are easier to maintain and more flexible. This chapter contains only an introduction to the use of inheritance for the purpose of improving program structures. More uses of inheritance and their benefits will be discussed in the following chapters.

Terms introduced in this chapter:

- inheritance, superclass (parent), subclass (child), is-a, inheritance hierarchy, abstract class, subtype substitution, polymorphic variable, type loss, cast
Exercise 10.16 Go back to the *lab-classes* project from Chapter 1. Add instructors to the project (every lab class can have many students and a single instructor). Use inheritance to avoid code duplication between students and instructors (both have a name, contact details, etc.).

Exercise 10.17 Draw an inheritance hierarchy representing parts of a computer system (processor, memory, disk drive, DVD drive, printer, scanner, keyboard, mouse, etc.).

Exercise 10.18 Look at the code below. You have four classes (O, X, T, and M) and a variable of each of these.

```java
O o;
X x;
T t;
M m;
```

The following assignments are all legal (assume that they all compile):

```java
m = t;
m = x;
o = t;
```

The following assignments are all illegal (they cause compiler errors):

```java
o = m;
o = x;
x = o;
```

What can you say about the relationships of these classes? Draw a class diagram.

Exercise 10.19 Draw an inheritance hierarchy of *AbstractList* and all its (direct and indirect) subclasses as they are defined in the Java standard library.
Main concepts discussed in this chapter:
- method polymorphism
- overriding
- static and dynamic type
- dynamic method lookup

Java constructs discussed in this chapter:
- super (in method), toString, protected, instanceof

The last chapter introduced the main concepts of inheritance by discussing the network example. While we have seen the foundations of inheritance, there are still numerous important details that we have not yet investigated. Inheritance is central to understanding and using object-oriented languages, and understanding it in detail is necessary to progress from here.

In this chapter, we shall continue to use the network example to explore the most important of the remaining issues surrounding inheritance and polymorphism.

### 11.1 The problem: network’s display method

When you experimented with the network examples in Chapter 10, you probably noticed that the second version—the one using inheritance—has a problem: the display method does not show all of a post’s data.

Let us look at an example. Assume that we create a MessagePost and a PhotoPost object with the following data:

The message post:

Leonardo da Vinci

Had a great idea this morning.
But now I forgot what it was. Something to do with flying . . .

40 seconds ago. 2 people like this.

No comments.
The photo post:

Alexander Graham Bell
[experiment.jpg] I think I might call this thing 'telephone'.
12 minutes ago. 4 people like this.
No comments.

If we enter these objects into the news feed\(^1\) and then invoke the first version of the news feed's `show` method (the one without inheritance), it prints

Leonardo da Vinci
Had a great idea this morning.
But now I forgot what it was. Something to do with flying ... 40 seconds ago - 2 people like this.
No comments.

Alexander Graham Bell
[experiment.jpg] I think I might call this thing 'telephone'.
12 minutes ago - 4 people like this.
No comments.

While the formatting isn't pretty (because, in the text terminal, we don't have formatting options available), all the information is there, and we can imagine how the `show` method might be adapted later to show the data in a nicer formatting in a different user interface.

Compare this with the second `network` version (with inheritance), which prints only

Leonardo da Vinci
40 seconds ago - 2 people like this.
No comments.

Alexander Graham Bell
12 minutes ago - 4 people like this.
No comments.

We note that the message post's text, as well as the photo post's image filename and caption, are missing. The reason for this is simple. The `display` method in this version is implemented in the `Post` class, not in `MessagePost` and `PhotoPost` (Figure 11.1). In the methods of `Post`, only the fields declared in `Post` are available. If we tried to access the `MessagePost`'s `message` field from `Post`'s `display` method, an error would be reported. This illustrates the important principle that inheritance is a one-way street: `MessagePost` inherits the fields of `Post`, but `Post` still does not know anything about fields in its subclasses.

\(^1\) The text for the message post is a two-line string. You can enter a multilne text into a string by using "\n" in the string for the line break.
11.2 Static type and dynamic type

Trying to solve the problem of developing a complete polymorphic `display` method leads us into a discussion of static and dynamic types and method lookup. But let us start at the beginning.

A first attempt to solve the display problem might be to move the `display` method to the subclasses (Figure 11.2). That way, because the method would now belong to the `MessagePost` and `PhotoPost` classes, it could access the specific fields of `MessagePost` and `PhotoPost`. It could also access the inherited fields by calling accessor methods defined in the `Post` class. That should enable it to display a complete set of information again. Try out this approach by completing Exercise 11.1.
Exercise 11.1 Open your last version of the network project. (You can use network-v2 if you do not have your own version yet.) Remove the display method from class Post and move it into the MessagePost and PhotoPost classes. Compile. What do you observe?

When we try to move the display method from Post to the subclasses, we notice that the project does not compile any more. There are two fundamental issues:

- We get errors in the MessagePost and PhotoPost classes, because we cannot access the superclass fields.
- We get an error in the NewsFeed class, because it cannot find the display method.

The reason for the first sort of error is that the fields in Post have private access, and so are inaccessible to any other class—including subclasses. Because we do not wish to break encapsulation and make these fields public, as was suggested above, the easiest way to solve this is to define public accessor methods for them. However, in Section 11.9, we shall introduce a further type of access designed specifically to support the superclass–subclass relationship.

The reason for the second sort of error requires a more detailed explanation, and this is explored in the next section.

11.2.1 Calling display from NewsFeed

First, we investigate the problem of calling the display method from NewsFeed. The relevant lines of code in the NewsFeed class are:

```
for(Post post : posts) {
    post.display();
    System.out.println();
}
```

The for-each statement retrieves each post from the collection; the first statement inside its body tries to invoke the display method on the post. The compiler informs us that it cannot find a display method for the post.

On the one hand, this seems logical; Post does not have a display method any more (see Figure 11.2).

On the other hand, it seems illogical and is annoying. We know that every Post object in the collection is in fact a MessagePost or a PhotoPost object, and both have display methods. This should mean that post.display() ought to work, because, whatever it is—MessagePost or PhotoPost—we know that it does have a display method.

To understand in detail why it does not work, we need to look more closely at types. Consider the following statement:

```
Car c1 = new Car();
```
We say that the type of \texttt{c1} is \texttt{Car}. Before we encountered inheritance, there was no need to distinguish whether by “type of \texttt{c1}” we meant “the type of the variable \texttt{c1}” or “the type of the object stored in \texttt{c1}.” It did not matter, because the type of the variable and the type of the object were always the same.

Now that we know about subtyping, we need to be more precise. Consider the following statement:

\[
\text{Vehicle v1 = new Car();}
\]

What is the type of \texttt{v1}? That depends on what precisely we mean by “type of \texttt{v1}.” The type of the variable \texttt{v1} is \texttt{Vehicle}; the type of the object stored in \texttt{v1} is \texttt{Car}. Through subtyping and substitution rules, we now have situations where the type of the variable and the type of the object stored in it are not exactly the same.

Let us introduce some terminology to make it easier to talk about this issue:

- We call the declared type of the variable the \textit{static type}, because it is declared in the source code—the static representation of the program.
- We call the type of the object stored in a variable the \textit{dynamic type}, because it depends on assignments at runtime—the dynamic behavior of the program.

Thus, looking at the explanations above, we can be more precise: the static type of \texttt{v1} is \texttt{Vehicle}, the dynamic type of \texttt{v1} is \texttt{Car}. We can now also rephrase our discussion about the call to the post's \texttt{display} method in the \texttt{NewsFeed} class. At the time of the call

\[
\text{post.display();}
\]

the static type of \texttt{post} is \texttt{Post}, while the dynamic type is either \texttt{MessagePost} or \texttt{PhotoPost} (Figure 11.3). We do not know which one of these it is, assuming that we have entered both \texttt{MessagePost} and \texttt{PhotoPost} objects into the feed.

The compiler reports an error because, for type checking, the static type is used. The dynamic type is often only known at runtime, so the compiler has no other choice but to use the static type if it wants to do any checks at compile time. The static type of \texttt{post} is \texttt{Post}, and \texttt{Post} does not have a \texttt{display} method. It makes no difference that all known subtypes of \texttt{Post} do have a \texttt{display} method. The behavior of the compiler is reasonable in this respect, because it has no guarantee that \textit{all} subclasses of \texttt{Post} will, indeed, define a \texttt{display} method, and this is impossible to check in practice.

In other words, to make this call work, class \texttt{Post} must have a \texttt{display} method, so we appear to be back to our original problem without having made any progress.
Exercise 11.2 In your network project, add a `display` method in class `Post` again. For now, write the method body with a single statement that prints out only the username. Then modify the `display` methods in `MessagePost` and `PhotoPost` so that the `MessagePost` version prints out only the message and the `PhotoPost` version prints only the caption. This removes the other errors encountered above (we shall come back to those below).

You should now have a situation corresponding to Figure 11.4, with `display` methods in three classes. Compile your project. (If there are errors, remove them. This design should work.)

Before executing, predict which of the `display` methods will get called if you execute the news feed’s `show` method.

Try it out. Enter a message post and a photo post into the news feed and call the news feed’s `show` method. Which `display` methods were executed? Was your prediction correct? Try to explain your observations.

**Figure 11.4**
Display, version 3: `display` method in subclasses and superclass

11.3 overriding

The next design we shall discuss is one where both the superclass and the subclasses have a `display` method (Figure 11.4). The header of all the `display` methods is exactly the same.

Code 11.1 shows the relevant details of the source code of all three classes. Class `Post` has a `display` method that prints out all the fields that are declared in `Post` (those common to message posts and photo posts), and the subclasses `MessagePost` and `PhotoPost` print out the fields specific to `MessagePost` and `PhotoPost` objects, respectively.
public class Post {

    ...

    public void display()
    {
        System.out.println(username);
        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.");
        }
    }

}

public class MessagePost extends Post {

    ...

    public void display()
    {
        System.out.println(message);
    }
}

public class PhotoPost extends Post {

    ...

    public void display()
    {
        System.out.println(" [" + filename + "]");
        System.out.println(" + caption");
    }
}

This design works a bit better. It compiles, and it can be executed, even though it is not perfect yet. An implementation of this design is provided in the project network-v3. (If you have done Exercise 11.2, you already have a similar implementation of this design in your own version.)
The technique we are using here is called overriding (sometimes it is also referred to as redefinition). Overriding is a situation where a method is defined in a superclass (method display in class Post in this example), and a method with exactly the same signature is defined in the subclass. The annotation @Override may be added before the version in the subclass to make it clear that a new version of an inherited method is being defined.

In this situation, objects of the subclass have two methods with the same name and header: one inherited from the superclass and one from the subclass. Which one will be executed when we call this method?

**Dynamic method lookup**

One surprising detail is what exactly is printed once we execute the news feed’s show method. If we again create and enter the objects described in Section 11.1, the output of the show method in our new version of the program is

```
Had a great idea this morning.
But now I forgot what it was. Something to do with flying . . .

[experiment.jpg]
I think I might call this thing 'telephone'.
```

We can see from this output that the display methods in MessagePost and in PhotoPost were executed, but not the one in Post.

This may seem strange at first. Our investigation in Section 11.2 has shown that the compiler insisted on a display method in class Post—methods in the subclasses were not enough. This experiment now shows that the method in class Post is then not executed at all, but the subclass methods are. In short:

- Type checking uses the static type, but at runtime, the methods from the dynamic type are executed.

This is a fairly important statement. To understand it better, we look in more detail at how methods are invoked. This procedure is known as method lookup, method binding, or method dispatch. We will use the term “method lookup” in this book.

We start with a simple method-lookup scenario. Assume that we have an object of a class PhotoPost stored in a variable v1 declared of type PhotoPost (Figure 11.5).
The `PhotoPost` class has a `display` method and no declared superclass. This is a very simple situation—there is no inheritance or polymorphism involved here. We then execute the statement

```java
v1.display();
```

When this statement executes, the `display` method is invoked in the following steps:

1. The variable `v1` is accessed.
2. The object stored in that variable is found (following the reference).
3. The class of the object is found (following the “instance of” reference).
4. The implementation of the `display` method is found in the class and executed.

This is all very straightforward and not surprising.

Next, we look at method lookup with inheritance. This scenario is similar, but this time the `PhotoPost` class has a superclass `Post`, and the `display` method is defined only in the superclass (Figure 11.6).

We execute the same statement. The method invocation then starts in a similar way: steps 1 through 3 from the previous scenario are executed again, but then it continues differently:

4. No `display` method is found in class `PhotoPost`.

5. Because no matching method was found, the superclass is searched for a matching method. If no method is found in the superclass, the next superclass (if it exists) is searched. This continues all the way up the inheritance hierarchy to the `Object` class, until a method is found. Note that at runtime, a matching method should definitely be found, or else the class would not have compiled.

6. In our example, the `display` method is found in class `Post`, and will be executed.

![Diagram](image-url)
This scenario illustrates how objects inherit methods. Any method found in a superclass can be invoked on a subclass object and will correctly be found and executed.

Next, we come to the most interesting scenario: method lookup with a polymorphic variable and method overriding (Figure 11.7). The scenario is again similar to the one before, but there are two changes:

- The declared type of the variable \texttt{v1} is now \texttt{Post}, not \texttt{PhotoPost}.
- The \texttt{display} method is defined in class \texttt{Post} and then redefined (overridden) in class \texttt{PhotoPost}.

This scenario is the most important one for understanding the behavior of our \textit{network} application, and in finding a solution to our \texttt{display} method problem.

The steps in which method execution takes place are exactly the same as steps 1 through 4 from scenario 1. Read them again.

Some observations are worth noting:

- No special lookup rules are used for method lookup in cases where the dynamic type is not equal to the static type. The behavior we observe is a result of the general rules.
- Which method is found first and executed is determined by the dynamic type, not the static type. In other words, the fact that the declared type of the variable \texttt{v1} is now \texttt{Post} does not have any effect. The instance we are dealing with is of class \texttt{PhotoPost}—that is all that matters.
- Overriding methods in subclasses take precedence over superclass methods. Because method lookup starts in the dynamic class of the instance (at the bottom of the inheritance hierarchy), the last redefinition of a method is found first, and this is the one that is executed.
- When a method is overridden, only the last version (the one lowest in the inheritance hierarchy) is executed. Versions of the same method in any superclasses are also not automatically executed.
This explains the behavior that we observe in our network project. Only the display methods in the subclasses (MessagePost and PhotoPost) are executed when posts are printed out, leading to incomplete listings. In the next section, we discuss how to fix this.

## 11.5 super call in methods

Now that we know in detail how overridden methods are executed, we can understand the solution to the problem. It is easy to see that what we would want to achieve is for every call to a display method of, say, a PhotoPost object, to result in both the display method of the Post class and that of the PhotoPost class being executed for the same object. Then all the details would be printed out. (A different solution will be discussed later in this chapter.)

This is, in fact, quite easy to achieve. We can simply use the super construct, which we have already encountered in the context of constructors in Chapter 10. Code 11.2 illustrates this idea with the display method of the PhotoPost class.

```java
public void display()
{
    super.display();
    System.out.println("[" + filename + "]");
    System.out.println("" + caption);
}
```

When display is now called on a PhotoPost object, initially the display method in the PhotoPost class will be invoked. As its first statement, this method will in turn invoke the display method of the superclass, which prints out the general post information. When control returns from the superclass method, the remaining statements of the subclass method print the distinctive fields of the PhotoPost class.

There are three details worth noting:

- Contrary to the case of super calls in constructors, the method name of the superclass method is explicitly stated. A super call in a method always has the form

  ```java
  super.method-name(parameters)
  ```

- The parameter list can, of course, be empty.

- Again, contrary to the rule for super calls in constructors, the super call in methods may occur anywhere within that method. It does not have to be the first statement.

- And contrary to the case of super calls in constructors, no automatic super call is generated and no super call is required; it is entirely optional. So the default behavior gives the effect of a subclass method completely hiding (i.e., overriding) the superclass version of the same method.

**Exercise 11.3** Modify your latest version of the network project to include the super call in the display method. Test it. Does it behave as expected? Do you see any problems with this solution?
It is worth reiterating what was illustrated in Exercise 10.6: that in the absence of method overriding, the non-private methods of a superclass are directly accessible from its subclasses without any special syntax. A super call only has to be made when it is necessary to access the superclass version of an overridden method.

If you completed Exercise 11.3, you will have noticed that this solution works, but is not perfect yet. It prints out all details, but in a different order from what we wanted. We will fix this last problem later in the chapter.

### Method polymorphism

What we have just discussed in the previous sections (Sections 11.2–11.5) is yet another form of polymorphism. It is what is known as polymorphic method dispatch (or method polymorphism for short).

Remember that a polymorphic variable is one that can store objects of varying types (every object variable in Java is potentially polymorphic). In a similar manner, Java method calls are polymorphic, because they may invoke different methods at different times. For instance, the statement

```java
post.display();
```

could invoke the `MessagePost`'s `display` method at one time and the `PhotoPost`'s `display` method at another, depending on the dynamic type of the `post` variable.

### Object methods: `toString`

In Chapter 10, we mentioned that the universal superclass, `Object`, implements some methods that are then part of all objects. The most interesting of these methods is `toString`, which we introduce here (if you are interested in more detail, you can look up the interface for `Object` in the standard library documentation).

**Exercise 11.4** Look up `toString` in the library documentation. What are its parameters? What is its return type?

The purpose of the `toString` method is to create a string representation of an object. This is useful for any objects that are ever to be textually represented in the user interface, but also helps for all other objects; they can then easily be printed out for debugging purposes, for instance.

The default implementation of `toString` in class `Object` cannot supply a great amount of detail. If, for example, we call `toString` on a `PhotoPost` object, we receive a string similar to this:

`PhotoPost@65c221c0`
The return value simply shows the object’s class name and a magic number.\(^2\)

**Exercise 11.5** You can easily try this out. Create an object of class `PhotoPost` in your project, and then invoke the `toString` method from the `Object` sub-menu in the object’s pop-up menu.

To make this method more useful, we would typically override it in our own classes. We can, for example, define the `Post`’s `display` method in terms of a call to its `toString` method. In this case, the `toString` method would not print out the details, but just create a string with the text. Code 11.3 shows the changed source code.

```java
public class Post
{
    ...
    public String toString()
    {
        String text = username + "\n" + timeString(timestamp);
        if(likes > 0)
        {
            text += "  - " + likes + " people like this.\n";
        }
        else
        {
            text += "\n";
        }

        if(comments.isEmpty())
        {
            return text + "  No comments.\n";
        }
        else
        {
            return text + "  " + comments.size() + " comment(s). Click here to view.\n";
        }
    }

    public void display()
    {
        System.out.println(toString());
    }
}
```

\(^2\) The magic number is in fact the memory address where the object is stored. It is not very useful except to establish identity. If this number is the same in two calls, we are looking at the same object. If it is different, we have two distinct objects.
public class MessagePost extends Post {

    ... 

    public String toString() {
        return super.toString() + message + "\n";
    }

    public void display() {
        System.out.println(toString());
    }
}

Ultimately, we would plan on removing the display methods completely from these classes. A great benefit of defining just a toString method is that we do not mandate in the Post classes what exactly is done with the description text. The original version always printed the text to the output terminal. Now, any client (e.g., the NewsFeed class) is free to do whatever it chooses with this text. It might show the text in a text area in a graphical user interface; save it to a file; send it over a network; show it in a web browser; or, as before, print it to the terminal.

The statement used in the client to print the post could now look as follows:

    System.out.println(post.toString());

In fact, the System.out.print and System.out.println methods are special in this respect: if the parameter to one of the methods is not a String object, then the method automatically invokes the object’s toString method. Thus we do not need to write the call explicitly and could instead write

    System.out.println(post);

Now consider the modified version of the show method of class NewsFeed shown in Code 11.4. In this version, we have removed the toString call. Would it compile and run correctly?

public class NewsFeed {

    Fields, constructors, and other methods omitted.

    /**
     * Show the news feed. Currently: print the news feed details to the
     * terminal. (To do: replace this later with display in web browser.)
     */
    public void show() {
        for(Post post : posts) {
            System.out.println(post);
        }
    }
}
In fact, the method *does* work as expected. If you can explain this example in detail, then you probably already have a good understanding of most of the concepts that we have introduced in this and the previous chapter! Here is a detailed explanation of the single `println` statement inside the loop.

- The for-each loop iterates through all posts and places them in a variable with the static type `Post`. The dynamic type is either `MessagePost` or `PhotoPost`.

- Because this object is being printed to `System.out` and it is not a `String`, its `toString` method is automatically invoked.

- Invoking this method is valid only because the class `Post` (the static type!) has a `toString` method. (Remember: Type checking is done with the static type. This call would not be allowed if class `Post` had no `toString` method. However, the `toString` method in class `Object` guarantees that this method is always available for any class.)

- The output appears properly with all details, because each possible dynamic type (`MessagePost` and `PhotoPost`) overrides the `toString` method and the dynamic method lookup ensures that the redefined method is executed.

The `toString` method is generally useful for debugging purposes. Often, it is very convenient if objects can easily be printed out in a sensible format. Most of the Java library classes override `toString` (all collections, for instance, can be printed out like this), and often it is a good idea to override this method for our classes as well.

### 11.8 Object equality: `equals` and `hashCode`

It is often necessary to determine whether two objects are "the same." The `Object` class defines two methods, `equals` and `hashCode`, that have a close link with determining similarity. We actually have to be careful when using phrases such as "the same"; this is because it can mean two quite different things when talking about objects. Sometimes we wish to know whether two different variables are referring to the same object. This is exactly what happens when an object variable is passed as a parameter to a method: there is only one object, but both the original variable and the parameter variable refer to it. The same thing happens when any object variable is assigned to another. These situations produce what is called *reference equality*. Reference equality is tested for using the `==` operator. So the following test will return `true` if both `var1` and `var2` are referring to the same object (or are both `null`), and `false` if they are referring to anything else:

```java
var1 == var2
```

Reference equality takes no account at all of the *contents* of the objects referred to, just whether there is one object referred to by two different variables or two distinct objects. That is why we also define *content equality*, as distinct from reference equality. A test for content equality asks whether two objects are the same internally—that is, whether the internal states of two objects are the same. This is why we rejected using reference equality for making string comparisons in Chapter 6.

What content equality between two particular objects means is something that is defined by the objects' class. This is where we make use of the `equals` method that every class inherits
from the **Object** superclass. If we need to define what it means for two objects to be equal according to their internal states, then we must override the **equals** method, which then allows us to write tests such as

```
var1.equals(var2)
```

This is because the **equals** method inherited from the **Object** class actually makes a test for reference equality. It looks something like this:

```
public boolean equals(Object obj) {
    return this == obj;
}
```

Because the **Object** class has no fields, there is no state to compare, and this method obviously cannot anticipate fields that might be present in subclasses.

The way to test for content equality between two objects is to test whether the values of their two sets of fields are equal. Notice, however, that the parameter of the **equals** method is of type **Object**, so a test of the fields will make sense only if we are comparing fields of the same type. This means that we first have to establish that the type of the object passed as a parameter is the same as that of the object it is being compared with. Here is how we might think of writing the method in the **Student** class of the **lab-classes** project from Chapter 1:

```
public boolean equals(Object obj) {
    if(this == obj) { // Reference equality.
        return true;
    }
    if(!(obj instanceof Student)) { // Not the same type.
        return false;
    }
    // Gain access to the other student's fields.
    Student other = (Student) obj;
    return name.equals(other.name) && id.equals(other.id) &&
           credits == other.credits;
}
```

The first test is just an efficiency improvement; if the object has been passed a reference to itself to compare against, then we know that content equality must be true. The second test makes sure that we are comparing two students. If not, then we decide that the two objects cannot be equal. Having established that we have another student, we use a cast and another variable of the right type so that we can access its details properly. Finally, we make use of the fact that the private elements of an object are directly accessible to an instance of the same class; this is essential in situations such as this one, because there will not necessarily be accessor methods defined for every private field in a class. Notice that we have consistently used content-equality tests rather than reference-equality tests on the object fields **name** and **id**.
It will not always be necessary to compare every field in two objects in order to establish that they are equal. For instance, if we know for certain that every Student is assigned a unique id, then we need not test the name and credits fields as well. It would then be possible to reduce the final statement above to

```java
return id.equals(other.id);
```

Whenever the equals method is overridden, the hashCode method should also be overridden. The hashCode method is used by data structures such as HashMap and HashSet to provide efficient placement and lookup of objects in these collections. Essentially, the hashCode method returns an integer value that represents an object. From the default implementation in Object, distinct objects have distinct hashCode values.

There is an important link between the equals and hashCode methods in that two objects that are the same as determined by a call to equals must return identical values from hashCode. This stipulation, or contract, can be found in the description of hashCode in the API documentation of the Object class. It is beyond the scope of this book to describe in detail a suitable technique for calculating hash codes, but we recommend that the interested reader see Joshua Bloch’s Effective Java, whose technique we use here. Essentially, an integer value should be computed making use of the values of the fields that are compared by the overridden equals method. Here is a hypothetical hashCode method that uses the values of an integer field called count and a String field called name to calculate the code:

```java
public int hashCode()
{
    int result = 17; // An arbitrary starting value.
    // Make the computed value depend on the order in which
    // the fields are processed.
    result = 37 * result + count;
    result = 37 * result + name.hashCode();
    return result;
}
```

## 11.9 Protected access

In Chapter 10, we noted that the rules on public and private visibility of class members apply between a subclass and its superclass, as well as between classes in different inheritance hierarchies. This can be somewhat restrictive, because the relationship between a superclass and its subclasses is clearly closer than it is with other classes. For this reason, object-oriented languages often define a level of access that lies between the complete restriction of private access and the full availability of public access. In Java, this is called protected access and is provided by the protected keyword as an alternative to public and private. Code 11.5 shows an example of a protected accessor method, which we could add to class Post.

---

3 Note that it is not essential that unequal objects always return distinct hash codes.

protected long getTimeStamp()
{
  return timestamp;
}

Protected access allows access to the fields or methods within a class itself and from all its subclasses, but not from other classes. The `getTimeStamp` method shown in Code 11.5 can be called from class `Post` or any subclasses, but not from other classes. Figure 11.8 illustrates this. The oval areas in the diagram show the group of classes that are able to access members in class `SomeClass`.

While protected access can be applied to any member of a class, it is usually reserved for methods and constructors. It is not usually applied to fields, because that would be a weakening of encapsulation. Wherever possible, mutable fields in superclasses should remain private. There are, however, occasional valid cases where direct access by subclasses is desirable. Inheritance represents a much closer form of coupling than does a normal client relationship.

Inheritance binds the classes closer together, and changing the superclass can more easily break the subclass. This should be taken into consideration when designing classes and their relationships.

---

5 In Java, this rule is not as clear-cut as described here, because Java includes an additional level of visibility, called package level, but with no associated keyword. We will not discuss this further, and it is more general to consider protected access as intended for the special relationship between superclass and subclass.
Exercise 11.6  The version of display shown in Code 11.2 produces the output shown in Figure 11.9. Reorder the statements in the method in your version of the network project so that it prints the details as shown in Figure 11.10.

Leonardo da Vinci
40 seconds ago - 2 people like this.
No comments.
Had a great idea this morning.
But now I forgot what it was. Something to do with flying...

Had a great idea this morning.
But now I forgot what it was. Something to do with flying...
Leonardo da Vinci
40 seconds ago - 2 people like this.
No comments.

Exercise 11.7  Having to use a superclass call in display is somewhat restrictive in the ways in which we can format the output, because it is dependent on the way the superclass formats its fields. Make any necessary changes to the Post class and to the display method of MessagePost so that it produces the output shown in Figure 11.11. Any changes you make to the Post class should be visible only to its subclasses. Hint: You could add protected accessors to do this.

Leonardo da Vinci
Had a great idea this morning.
But now I forgot what it was. Something to do with flying...
40 seconds ago - 2 people like this.
No comments.

The instanceof operator

One of the consequences of the introduction of inheritance into the network project has been that the NewsFeed class knows only about Post objects and cannot distinguish between message posts and photo posts. This has allowed all types of posts to be stored in a single list.
However, suppose that we wish to retrieve just the message posts or just the photo posts from the list; how would we do that? Or perhaps we wish to look for a message by a particular author? That is not a problem if the Post class defines a getAuthor method, but this will find both message and photo posts. Will it matter which type is returned?

There are occasions when we need to rediscover the distinctive dynamic type of an object rather than dealing with a shared supertype. For this, Java provides the instanceof operator. The instanceof operator tests whether a given object is, directly or indirectly, an instance of a given class. The test

\[
\text{obj instanceof MyClass}
\]

returns true if the dynamic type of obj is MyClass or any subclass of MyClass. The left operand is always an object reference, and the right operand is always the name of a class. So

\[
\text{post instanceof MessagePost}
\]

returns true only if post is a MessagePost, as opposed to a PhotoPost, for instance.

Use of the instanceof operator is often followed immediately by a cast of the object reference to the identified type. For instance, here is some code to identify all of the message posts in a list of posts and to store them in a separate list.

```java
ArrayList<MessagePost> messages = new ArrayList<>();
for(Post post : posts) {
    if(post instanceof MessagePost) {
        messages.add((MessagePost) post);
    }
}
```

It should be clear that the cast here does not alter the post object in any way, because we have just established that it already is a MessagePost object.

### 11.11 Another example of inheritance with overriding

To discuss another example of a similar use of inheritance, we go back to a project from Chapter 8: the zuul project. In the world-of-zuul game, we used a set of Room objects to create a scene for a simple game. Exercise 8.45 suggested that you implement a transporter room (a room that beamds you to a random location in the game if you try to enter or leave it). We revisit this exercise here, because its solution can greatly benefit from inheritance. If you don’t remember this project well, have a quick read through Chapter 8 again, or look at your own zuul project.

There is no single solution to this task. Many different solutions are possible and can be made to work. Some are better than others—more elegant, easier to read, and easier to maintain and to extend.

Assume that we want to implement this task so that the player is automatically transported to a random room when she tries to leave the magic transporter room. The most straightforward solution that comes to mind first for many people is to deal with this in the Game class, which implements the player’s commands. One of the commands is go, which is
implemented in the `goRoom` method. In this method, we used the following statement as the central section of code:

```java
nextRoom = currentRoom.getExit(direction);
```

This statement retrieves from the current room the neighboring room in the direction we want to go. To add our magic transportation, we could modify this in a form similar to the following:

```java
if(currentRoom.getName().equals("Transporter room")) {
    nextRoom = getRandomRoom();
} else {
    nextRoom = currentRoom.getExit(direction);
}
```

The idea here is simple: we just check whether we are in the transporter room. If we are, then we find the next room by getting a random room (of course, we have to implement the `getRandomRoom` method somehow); otherwise, we just do the same as before.

While this solution works, it has several drawbacks. The first is that it is a bad idea to use text strings, such as the room's name, to identify the room. Imagine that someone wanted to translate your game into another language—say, to German. They might change the names of the rooms—"Transporter room" becomes "Transporterraum"—and suddenly the game does not work any more! This is a clear case of a maintainability problem.

The second solution, which is slightly better, would be to use an instance variable instead of the room's name to identify the transporter room. Similar to this:

```java
if(currentRoom == transporterRoom) {
    nextRoom = getRandomRoom();
} else {
    nextRoom = currentRoom.getExit(direction);
}
```

This time, we assume that we have an instance variable `transporterRoom` of class `Room`, where we store the reference to our transporter room. Now the check is independent of the room's name. That is a bit better.

There is still a case for further improvement, though. We can understand the shortcomings of this solution when we think about another maintenance change. Imagine that we want to add two more transporter rooms, so that our game has three different transporter locations.

A very nice aspect of our existing design was that we could set up the floor plan in a single spot, and the rest of the game was completely independent of it. We could easily change the layout of the rooms, and everything would still work—high score for maintainability! With our current solution, though, this is broken. If we add two new transporter rooms, we have to add two more instance variables or an array (to store references to those rooms), and we have to modify our `goRoom` method to add a check for those rooms. In terms of easy changeability, we have gone backwards.

---

6 Make sure that you understand why a test for reference equality is the most appropriate here.
The question, therefore, is: Can we find a solution that does not require a change to the command implementation each time we add a new transporter room? Following is our idea:

We can add a method `isTransporterRoom` in the `Room` class. This way, the `Game` object does not need to remember all the transporter rooms—the rooms themselves do. When rooms are created, they could receive a boolean flag indicating whether a given room is a transporter room. The `goRoom` method could then use the following code segment:

```java
if(currentRoom.isTransporterRoom()) {
    nextRoom = getRandomRoom();
} else {
    nextRoom = currentRoom.getExit(direction);
}
```

Now we can add as many transporter rooms as we like; there is no need for any more changes to the `Game` class. However, the `Room` class has an extra field whose value is really needed only because of the nature of one or two of the instances. Special-case code such as this is a typical indicator of a weakness in class design. This approach also does not scale well should we decide to introduce further sorts of special rooms, each requiring its own flag field and accessor method.?

With inheritance, we can do better and implement a solution that is even more flexible than this one. We can implement a class `TransporterRoom` as a subclass of class `Room`. In this new class, we override the `getExit` method and change its implementation so that it returns a random room:

```java
public class TransporterRoom extends Room {
    /**
     * Return a random room, independent of the direction parameter.
     * @param direction Ignored.
     * @return A random room.
     */
    public Room getExit(String direction) {
        return findRandomRoom();
    }

    /*
     * Choose a random room.
     * @return A random room.
     */
    private Room findRandomRoom() {
        ... // implementation omitted
    }
}
```

7 We might also think of using `instanceof`, but the point here is that none of these ideas is the best.
The elegance of this solution lies in the fact that no change at all is needed in either the original Game or Room classes! We can simply add this class to the existing game, and the goRoom method will continue to work as it is. Adding the creation of a TransporterRoom to the setup of the floor plan is (almost) enough to make it work. Note, too, that the new class does not need a flag to indicate its special nature—its very type and distinctive behavior supply that information.

Because TransporterRoom is a subclass of Room, it can be used everywhere a Room object is expected. Thus, it can be used as a neighboring room for another room or be held in the Game object as the current room.

What we have left out, of course, is the implementation of the findRandomRoom method. In reality, this is probably better done in a separate class (say RoomRandomizer) than in the TransporterRoom class itself. We leave this as an exercise for the reader.

**Exercise 11.8** Implement a transporter room with inheritance in your version of the zuul project.

**Exercise 11.9** Discuss how inheritance could be used in the zuul project to implement a player and a monster class.

**Exercise 11.10** Could (or should) inheritance be used to create an inheritance relationship (super-, sub-, or sibling class) between a character in the game and an item?

## 11.12 Summary

When we deal with classes with subclasses and polymorphic variables, we have to distinguish between the static and dynamic type of a variable. The static type is the declared type, while the dynamic type is the type of the object currently stored in the variable.

Type checking is done by the compiler using the static type, whereas at runtime method lookup uses the dynamic type. This enables us to create very flexible structures by overriding methods. Even when using a supertype variable to make a method call, overriding enables us to ensure that specialized methods are invoked for every particular subtype. This ensures that objects of different classes can react distinctly to the same method call.

When implementing overriding methods, the super keyword can be used to invoke the superclass version of the method. If fields or methods are declared with the protected access modifier, subclasses are allowed to access them, but other classes are not.

Terms introduced in this chapter:

- static type
- dynamic type
- overriding
- redefinition
- method lookup
- method dispatch
- method polymorphism
- protected
Exercise 11.11 Assume that you see the following lines of code:

```java
Device dev = new Printer();
dev.getName();
```

*Printer* is a subclass of *Device*. Which of these classes must have a definition of method *getName* for this code to compile?

Exercise 11.12 In the same situation as in the previous exercise, if both classes have an implementation of *getName*, which one will be executed?

Exercise 11.13 Assume that you write a class *Student* that does not have a declared superclass. You do not write a *toString* method. Consider the following lines of code:

```java
Student st = new Student();
String s = st.toString();
```

Will these lines compile? If so, what exactly will happen when you try to execute?

Exercise 11.14 In the same situation as before (class *Student*, no *toString* method), will the following lines compile? Why?

```java
Student st = new Student();
System.out.println(st);
```
Exercise 11.15 Assume that your class Student overrides toString so that it returns the student's name. You now have a list of students. Will the following code compile? If not, why not? If yes, what will it print? Explain in detail what happens.

```java
for (Object st : myList) {
    System.out.println(st);
}
```

Exercise 11.16 Write a few lines of code that result in a situation where a variable x has the static type T and the dynamic type D.
Main concepts discussed in this chapter:

- abstract classes
- multiple inheritance
- interfaces

Java constructs discussed in this chapter:

abstract, implements, interface

In this chapter, we examine further inheritance-related techniques that can be used to enhance class structures and improve maintainability and extensibility. These techniques introduce an improved method of representation of abstractions in object-oriented programs.

The previous two chapters have discussed the most important aspects of inheritance in application design, but several more advanced uses and problems have been ignored so far. We will now complete the picture with a more advanced example.

The project we use for this chapter is a simulation. We use it to discuss inheritance again and see that we run into some new problems. Abstract classes and interfaces are then introduced to deal with these problems.

12.1 Simulations

Computers are frequently used to simulate real systems. These include systems that model traffic flows in a city, forecast weather, simulate the spread of infection, analyze the stock market, do environmental simulations, and much more. In fact, many of the most powerful computers in the world are used for running some sort of simulation.

When creating a computer simulation, we try to model the behavior of a subset of the real world in a software model. Every simulation is necessarily a simplification of the real thing. Deciding which details to leave out and which to include is often a challenging task. The more detailed a simulation is, the more accurate it may be in forecasting the behavior of the real system. But more detail increases the complexity of the model and the requirements for both more computing power and more programmer time. A well-known example is weather forecasting: climate models in weather simulations have been
improved by adding more and more detail over the last few decades. As a result, weather forecasts have improved significantly in accuracy (but are far from perfect, as we all have experienced at some time). Much of this improvement has been made possible through advances in computer technology.

The benefit of simulations is that we can undertake experiments that we could not do with the real system, either because we have no control over the real thing (for instance, the weather) or because it is too costly, too dangerous, or irreversible in case of disaster. We can use the simulation to investigate the behavior of the system under certain circumstances or to investigate “what if” questions.

An example of the use of environmental simulations is to try to predict the effects of human activity on natural habitats. Consider the case of a national park containing endangered species and a proposal to build a freeway through the middle of it, completely separating the two halves. The supporters of the freeway proposal claim that splitting the park in half will lead to little actual land loss and make no difference to the animals in it, but environmentalists claim otherwise. How can we tell what the effect is likely to be without building the freeway? Simulation is one option. An essential question in all cases of this kind will be, of course, “How good is the simulation?” One can “prove” just about anything with an ill-designed simulation. Gaining trust in it through controlled experiments will be essential.

The issue in this particular case boils down to whether it is significant for the survival of a species to have a single, connected habitat area, or whether two disjoint areas (with the same total size as the other) are just as good. Rather than building the freeway first and then observing what happens, we will try to simulate the effect in order to make a well-informed decision.

Our simulation will necessarily be simpler than the scenario we have described, because we are using it mainly to illustrate new features of object-oriented design and implementation. Therefore, it will not have the potential to simulate accurately many aspects of nature, but some of the simulation’s characteristics are nonetheless interesting. In particular, it will demonstrate the structure of typical simulations. In addition, its accuracy may surprise you; it would be a mistake to equate greater complexity with greater accuracy. It is often the case that a simplified model of something can provide greater insight and understanding than a more complex one, from which it is often difficult to isolate the underlying mechanisms—or even be sure that the model is valid.

### 12.2 The foxes-and-rabbits simulation

The simulation scenario we have chosen to work with in this chapter uses the freeway example from above as its basis. It involves tracking populations of foxes and rabbits within an enclosed area. This is just one particular example of what are known as predator–prey

---

1 In this particular case, by the way, size does matter: the size of a natural park has a significant impact on its usefulness as a habitat for animals.
Simulations. Such simulations are often used to model the variation in population sizes that result from a predator species feeding on a prey species. A delicate balance exists between such species. A large population of prey will potentially provide plenty of food for a small population of predators. However, too many predators could kill off all the prey and leave the hunters with nothing to eat. Population sizes could also be affected by the size and nature of the environment. For instance, a small, enclosed environment could lead to overcrowding and make it easy for the predators to locate their prey, or a polluted environment could reduce the stock of prey and prevent even a modest population of predators from surviving. Because predators in one context are often prey for other species (think of cats, birds, and worms, for instance), loss of one part of the food chain can have dramatic effects on the survival of other parts.

As we have done in previous chapters, we will start with a version of an application that works perfectly well from a user’s point of view, but whose internal view is not so good when judged by the principles of good object-oriented design and implementation. We will use this base version to develop several improved versions that progressively introduce new abstraction techniques.

One particular problem that we wish to address in the base version is that it does not make good use of the inheritance techniques that were introduced in Chapter 10. However, we will start by examining the mechanism of the simulation, without being too critical of its implementation. Once we understand how it works, we shall be in a good position to make some improvements.

**Predator-prey modeling** There is a long history of trying to model predator-prey relationships mathematically before the invention of the computer, because they have economic, as well as environmental, importance. For instance, mathematical models were used in the early twentieth century to explain variations in the level of fish stocks in the Adriatic Sea as a side effect of World War I. To find out more about the background of this topic, and perhaps gain an understanding of population dynamics, do a web search for the Lotka-Volterra model.

### 12.2.1 The foxes-and-rabbits project

Open the `foxes-and-rabbits-v1` project. The class diagram is shown in Figure 12.1.

The main classes we will focus on in our discussion are *Simulator*, *Fox*, and *Rabbit*. The *Fox* and *Rabbit* classes provide simple models of the behavior of a predator and prey, respectively. In this particular implementation, we have not tried to provide an accurate biological model of real foxes and rabbits; rather, we are simply trying to illustrate the principles of typical predator-prey simulations. Our main concerns will be on the aspects that most affect population size: birth, death, and food supply.

The *Simulator* class is responsible for creating the initial state of the simulation, then controlling and executing it. The basic idea is simple: the simulator holds collections of foxes and rabbits, and it repeatedly gives those animals an opportunity to live through one
Figure 12.1
Class diagram of the foxes-and-rabbits project

step\(^2\) of their life cycle. At each step, each fox and each rabbit is allowed to carry out the actions that characterize their behaviors. After each step (when all the animals have had the chance to act), the new current state of the field is displayed on screen.

We can summarize the purpose of the remaining classes as follows:

- **Field** represents a two-dimensional enclosed field. The field is composed of a fixed number of locations, which are arranged in rows and columns. At most, one animal may occupy a single location within the field. Each field location can hold an animal, or it can be empty.

- **Location** represents a two-dimensional position within the field, specified by a row and a column value.

- These five classes together (Simulator, Fox, Rabbit, Field, and Location) provide the model for the simulation. They completely determine the simulation behavior.

\(^2\) We won’t define how much time a “step” actually represents. In practice, this has to be decided by a combination of such things as what we are trying to discover, what events we are simulating, and how much real time is available to run the simulation.
The Randomizer class provides us with a degree of control over random aspects of the simulation, such as when new animals are born.

The classes SimulatorView, FieldStats, and Counter provide a graphical display of the simulation. The display shows an image of the field and counters for each species (the current number of rabbits and foxes).

SimulatorView provides a visualization of the state of the field. An example can be seen in Figure 12.2.

FieldStats provides to the visualization counts of the numbers of foxes and rabbits in the field.

A Counter stores a current count for one type of animal to assist with the counting.

Try the following exercises to gain an understanding of how the simulation operates before reading about its implementation.

Exercise 12.1 Create a Simulator object, using the constructor without parameters, and you should see an initial state of the simulation similar to that in Figure 12.2. The more numerous rectangles represent the rabbits. Does the number of foxes change if you call the simulateOneStep method just once?

Exercise 12.2 Does the number of foxes change on every step? What natural processes do you think we are modeling that cause the number of foxes to increase or decrease?

Exercise 12.3 Call the simulate method with a parameter to run the simulation continuously for a significant number of steps, such as 50 or 100. Do the numbers of foxes and rabbits increase or decrease at similar rates?
Exercise 12.4 What changes do you notice if you run the simulation for a much longer time, say for 4,000 steps? You can use the runLongSimulation method to do this.

Exercise 12.5 Use the reset method to create a new starting state for the simulation, and then run it again. Is an identical simulation run this time? If not, do you see broadly similar patterns emerging anyway?

Exercise 12.6 If you run a simulation for long enough, do all of the foxes or all of the rabbits ever die off completely? If so, can you pinpoint any reasons why that might be occurring?

Exercise 12.7 In the source code of the Simulator class, find the simulate method. In its body, you will see a call to a delay method that is commented out. Uncomment this call and run the simulation. Experiment with different delays so you can observe the simulation behavior more clearly. At the end, leave it in a state that makes the simulation look useful on your computer.

Exercise 12.8 Make a note of the numbers of foxes and rabbits at each of the first few steps, and at the end of a long run. It will be useful to have a record of these when we come to make changes later on and perform regression testing.

Exercise 12.9 After having run the simulation for a while, call the static reset method of the Randomizer class, and then the reset method of the Simulator object. Now run the first few steps again, and you should see the original simulation repeated. Take a look at the code of the Randomizer class to see if you can work out why this might be. You might need to look at the API for the java.util.Random class to help you with this.

Exercise 12.10 Check to see that setting the useShared field in Randomizer to false breaks the repeatability of the simulations seen in Exercise 12.9. Be sure to restore it to true afterwards, because repeatability will be an important element in later testing.

Now that we have a broad, external understanding of what this project does, we will look in detail at the implementation of the Rabbit, Fox, and Simulator classes.

12.2.2 The Rabbit class

The source code of the Rabbit class is shown in Code 12.1.

The Rabbit class contains a number of class variables that define configuration settings that are common to all rabbits. These include values for the maximum age to which a rabbit can live (defined as a number of simulation steps) and the maximum number of offspring
public class Rabbit
{
    // Characteristics shared by all rabbits (class variables).

    // The age at which a rabbit can start to breed.
    private static final int BREEDING_AGE = 5;
    // The age to which a rabbit can live.
    private static final int MAX_AGE = 40;
    // The likelihood of a rabbit breeding.
    private static final double BREEDING_PROBABILITY = 0.12;
    // The maximum number of births.
    private static final int MAX_LITTER_SIZE = 4;
    // A shared random number generator to control breeding.
    private static final Random rand = Randomizer.getRandom();

    // Individual characteristics (instance fields).

    // The rabbit's age.
    private int age;
    // Whether the rabbit is alive or not.
    private boolean alive;
    // The rabbit's position.
    private Location location;
    // The field occupied.
    private Field field;

    /**
     * Create a new rabbit. A rabbit may be created with age zero (a new born) or with a random age.
     * @param randomAge If true, the rabbit will have a random age.
     * @param field The field currently occupied.
     * @param location The location within the field.
     */
    public Rabbit(boolean randomAge, Field field, Location location)
    {
        // Body of constructor omitted.
    }

    /**
     * This is what the rabbit does most of the time - it runs around. Sometimes it will breed or die of old age.
     * @param newRabbits A list to return newly born rabbits.
     */
    public void run(List<Rabbit> newRabbits)
    {
        incrementAge();
        if(alive) {
            giveBirth(newRabbits);
            // Try to move into a free location.
            Location newLocation = field.freeAdjacentLocation(location);
            if(newLocation != null) {
                setLocation(newLocation);
            }
        }
    }
else {
    // Overcrowding.
    setDead();
}

/**
 * Indicate that the rabbit is no longer alive.
 * It is removed from the field.
 */
public void setDead()
{
    alive = false;
    if(location != null) {
        field.clear(location);
        location = null;
        field = null;
    }
}

/**
 * Increase the age.
 * This could result in the rabbit's death.
 */
private void incrementAge()
{
    age++;
    if(age > MAX_AGE) {
        setDead();
    }
}

/**
 * Check whether or not this rabbit is to give birth at this step.
 * New births will be made into free adjacent locations.
 * @param newRabbits A list to return newly born rabbits.
 */
private void giveBirth(List<Rabbit> newRabbits)
{
    // New rabbits are born into adjacent locations.
    // Get a list of adjacent free locations.
    List<Location> free = field.getFreeAdjacentLocations(location);
    int births = breed();
    for(int b = 0; b < births && free.size() > 0; b++) {
        Location loc = free.remove(0);
        Rabbit young = new Rabbit(false, field, loc);
        newRabbits.add(young);
    }
}

/**
 * Generate a number representing the number of births,
 * if it can breed.
 * @return The number of births (may be zero).
 */
private int breed()
{
    int births = 0;
    if(canBreed() && rand.nextDouble() <= BREEDING_PROBABILITY) {
        births = rand.nextInt(MAX_LITTER_SIZE) + 1;
    }
    return births;
}

Other methods omitted.

it can produce at any one step. Centralized control of random aspects of the simulation is provided through a single, shared Random object supplied by the Randomizer class. This is what makes possible the repeatability seen in Exercise 12.9. In addition, each individual rabbit has four instance variables that describe its state: its age as a number of steps, whether it is still alive, and its location in a particular field.

Exercise 12.11 Do you feel that omitting gender as an attribute in the Rabbit class is likely to lead to an inaccurate simulation? Write down the arguments for and against including it.

Exercise 12.12 Are there other simplifications that you feel are present in our implementation of the Rabbit class, compared with real life? Discuss whether these could have a significant impact on the accuracy of the simulation.

Exercise 12.13 Experiment with the effects of altering some or all of the values of the class variables in the Rabbit class. For instance, what effect does it have on the populations if the breeding probability of rabbits is much higher or much lower than it currently is?

A rabbit’s behavior is defined in its run method, which in turn uses the giveBirth and incrementAge methods and implements the rabbit’s movement. At each simulation step, the run method will be called and a rabbit will increase its age; if old enough, it might also breed, and it will then try to move. Both the movement and the breeding behaviors have random components. The direction in which the rabbit moves is randomly chosen, and breeding occurs randomly, controlled by the class variable BREEDING_PROBABILITY.

You can already see some of the simplifications that we have made in our model of rabbits: there is no attempt to distinguish males from females, for instance, and a rabbit could potentially give birth to a new litter at every simulation step once it is old enough.
12.2.3 The Fox class

There is a lot of similarity between the Fox and the Rabbit classes, so only the distinctive elements of Fox are shown in Code 12.2.

Code 12.2

The Fox class

```java
public class Fox
{
    // Characteristics shared by all foxes (class variables).
    private static final int RABBIT_FOOD_VALUE = 9;

    // The food value of a single rabbit. In effect, this is the
    // number of steps a fox can go before it has to eat again.
    private int foodValue;

    // Individual characteristics (instance fields).
    private int age;
    private boolean alive;
    private Location location;
    private Field field;

    // The fox's age.
    public void incrementAge()
    {
        // Body of constructor omitted.
    }

    // This is what the fox does most of the time: it hunts for
    // rabbits. In the process, it might breed, die of hunger,
    // or die of old age.
    public void hunt(List<Fox> newFoxes)
    {
        incrementAge();
        incrementHunger();
    }

    // Whether the fox is alive or not.
    public void incrementHunger()
    {
        // Other static fields omitted.
    }
}
```

Import statements and class comment omitted.
if(alive) {
    giveBirth(newFoxes);
    // Move towards a source of food if found.
    Location newLocation = findFood();
    if(newLocation == null) {
        // No food found - try to move to a free location.
        newLocation = field.freeAdjacentLocation(location);
    }
    // See if it was possible to move.
    if(newLocation != null) {
        setLocation(newLocation);
    } else {
        // Overcrowding.
        setDead();
    }
}

/**
 * Look for rabbits adjacent to the current location.
 * Only the first live rabbit is eaten.
 * @return Where food was found, or null if it wasn't.
 */
private Location findFood()
{
    List<Location> adjacent = field.adjacentLocations(location);
    Iterator<Location> it = adjacent.iterator();
    while(it.hasNext()) {
        Location where = it.next();
        Object animal = field.getObjectAt(where);
        if(animal instanceof Rabbit) {
            Rabbit rabbit = (Rabbit) animal;
            if(rabbit.isAlive()) {
                rabbit.setDead();
                foodLevel = RABBIT_FOOD_VALUE;
                return where;
            }
        }
    }
    return null;
}

Other methods omitted.

For foxes, the hunt method is invoked at each step and defines their behavior. In addition to aging and possibly breeding at each step, a fox searches for food (using findFood). If it is able to find a rabbit in an adjacent location, then the rabbit is killed and the fox's food level is increased. As with rabbits, a fox that is unable to move is considered dead through overcrowding.
Exercise 12.14 As you did for rabbits, assess the degree to which we have simplified the model of foxes and evaluate whether you feel the simplifications are likely to lead to an inaccurate simulation.

Exercise 12.15 Does increasing the maximum age for foxes lead to a significantly higher numbers of foxes throughout a simulation, or is the rabbit population more likely to be reduced to zero as a result?

Exercise 12.16 Experiment with different combinations of settings (breeding age, maximum age, breeding probability, litter size, etc.) for foxes and rabbits. Do species always disappear completely in some configurations? Are there configurations that are stable—i.e., that produce a balance of numbers for a significant length of time?

Exercise 12.17 Experiment with different sizes of fields. (You can do this by using the second constructor of Simulator.) Does the size of the field affect the likelihood of species surviving?

Exercise 12.18 Compare the results of running a simulation with a single large field, and two simulations with fields that each have half the area of the single field. This models something close to splitting an area in half with a freeway. Do you notice any significant differences in the population dynamics between the two scenarios?

Exercise 12.19 Repeat the investigations of the previous exercise, but vary the proportions of the two smaller fields. For instance, try three-quarters and one quarter, or two-thirds and one-third. Does it matter at all how the single field is split?

Exercise 12.20 Currently, a fox will eat at most one rabbit at each step. Modify the findFood method so that rabbits in all adjacent locations are eaten at a single step. Assess the impact of this change on the results of the simulation. Note that the findFood method currently returns the location of the single rabbit that is eaten, so you will need to return the location of one of the eaten rabbits in your version. However, don’t forget to return null if there are no rabbits to eat.

Exercise 12.21 Following from the previous exercise, if a fox eats multiple rabbits at a single step, there are several different possibilities as to how we can model its food level. If we add all the rabbit’s food values, the fox will have a very high food level, making it unlikely to die of hunger for a very long time. Alternatively, we could impose a ceiling on the fox’s food level. This models the effect of a predator that kills prey regardless of whether it is hungry or not. Assess the impacts on the resulting simulation of implementing this choice.
Exercise 12.22  Challenge exercise Given the random elements in the simulation, argue why the population numbers in an apparently stable simulation could ultimately collapse.

12.2.4 The Simulator class: setup

The Simulator class is the central part of the application that coordinates all the other pieces. Code 12.3 illustrates some of its main features.

Code 12.3

Part of the Simulator class

```java
public class Simulator
{
    // Static variables omitted.

    // Lists of animals in the field.
    private List<Rabbit> rabbits;
    private List<Fox> foxes;
    // The current state of the field.
    private Field field;
    // The current step of the simulation.
    private int step;
    // A graphical view of the simulation.
    private SimulatorView view;

    /**
     * Create a simulation field with the given size.
     * @param depth Depth of the field. Must be greater than zero.
     * @param width Width of the field. Must be greater than zero.
     */
    public Simulator(int depth, int width)
    {
        if(width <= 0 || depth <= 0)
        {
            System.out.println("The dimensions must be >= zero.");
            System.out.println("Using default values.");
            depth = DEFAULT_DEPTH;
            width = DEFAULT_WIDTH;
        }

        rabbits = new ArrayList<>();
        foxes = new ArrayList<>();
        field = new Field(depth, width);

        // Create a view of the state of each location in the field.
        view = new SimulatorView(depth, width);
        view.setColor(Rabbit.class, Color.ORANGE);
        view.setColor(Fox.class, Color.BLUE);

        // Setup a valid starting point
        reset();
    }
```
/**
 * Run the simulation for the given number of steps.
 * Stop before the given number of steps if it ceases to be viable.
 * @param numSteps The number of steps to run for.
 */
public void simulate(int numSteps)
{
    for(int step=1; step <= numSteps && view.isViable(field); step++) {
        simulateOneStep();
        // delay(60);  // uncomment this to run more slowly
    }
}

/**
 * Run the simulation from its current state for a single step. Iterate
 * over the whole field updating the state of each fox and rabbit.
 */
public void simulateOneStep()
{
    // Method body omitted
}

/**
 * Reset the simulation to a starting position.
 */
public void reset()
{
    step = 0;
    rabbits.clear();
    foxes.clear();
    populate();

    // Show the starting state in the view.
    view.showStatus(step, field);
}

/**
 * Randomly populate the field with foxes and rabbits.
 */
private void populate()
{
    Random rand = Randomizer.getRandom();
    field.clear();
    for(int row = 0; row < field.getDepth(); row++) {
        for(int col = 0; col < field.getWidth(); col++) {
            if(rand.nextDouble() <= FOX_CREATION_PROBABILITY) {
                Location location = new Location(row, col);
                Fox fox = new Fox(true, field, location);
                foxes.add(fox);
            } else if(rand.nextDouble() <= RABBIT_CREATION_PROBABILITY) {
                Location location = new Location(row, col);
                Rabbit rabbit = new Rabbit(true, field, location);
                rabbits.add(rabbit);
            }
        }
    }
}
12.2 The foxes-and-rabbits simulation

The **Simulator** has three important parts: its constructor, the `populate` method, and the `simulateOneStep` method. (The body of `simulateOneStep` is shown in Code 12.4.)

When a **Simulator** object is created, all other parts of the simulation are constructed by it (the field, the lists to hold the different types of animals, and the graphical interface). Once all these have been set up, the simulator’s `populate` method is called (indirectly, via the `reset` method) to create the initial populations. Different probabilities are used to decide whether a particular location will contain one of these animals. Note that animals created at the start of the simulation are given a random initial age. This serves two purposes:

1. It represents more accurately a mixed-age population that should be the normal state of the simulation.
2. If all animals were to start with an age of zero, no new animals would be created until the initial population had reached their respective breeding ages. With foxes eating rabbits regardless of the fox’s age, there is a risk that either the rabbit population will be killed off before it has a chance to reproduce, or that the fox population will die of hunger.

**Exercise 12.23** Modify the `populate` method of **Simulator** to determine whether setting an initial age of zero for foxes and rabbits is always catastrophic. Make sure that you run it a sufficient number of times—with different initial states, of course!

**Exercise 12.24** If an initial random age is set for rabbits but not foxes, the rabbit population will tend to grow large while the fox population remains very small. Once the foxes do become old enough to breed, does the simulation tend to behave again like the original version? What does this suggest about the relative sizes of the initial populations and their impact on the outcome of the simulation?

### 12.2.5 The **Simulator** class: a simulation step

The central part of the **Simulator** class is the `simulateOneStep` method shown in Code 12.4. It uses separate loops to let each type of animal move (and possibly breed or do whatever it is programmed to do). Because each animal can give birth to new animals, lists for these to be stored in are passed as parameters to the `hunt` and `run` methods of **Fox** and **Rabbit**. The newly born animals are then added to the master lists at the end of the step. Running longer simulations is trivial. To do this, the `simulateOneStep` method is called repeatedly in a simple loop.
public void simulateOneStep()
{
    step++;

    // Provide space for newborn rabbits.
    List<Rabbit> newRabbits = new ArrayList<>();
    // Let all rabbits act.
    for(Iterator<Rabbit> it = rabbits.iterator(); it.hasNext(); ) {
        Rabbit rabbit = it.next();
        rabbit.run(newRabbits);
        if(! rabbit.isAlive()) {
            it.remove();
        }
    }

    // Provide space for newborn foxes.
    List<Fox> newFoxes = new ArrayList<>();
    // Let all foxes act.
    for(Iterator<Fox> it = foxes.iterator(); it.hasNext(); ) {
        Fox fox = it.next();
        fox.hunt(newFoxes);
        if(! fox.isAlive()) {
            it.remove();
        }
    }

    // Add the newly born foxes and rabbits to the main lists.
    rabbits.addAll(newRabbits);
    foxes.addAll(newFoxes);

    view.showStatus(step, field);
}

In order to let each animal act, the simulator holds separate lists of the different types of animals. Here, we make no use of inheritance, and the situation is reminiscent of the first version of the network project introduced in Chapter 10.

Exercise 12.25 Each animal is always held in two different data structures: the Field and the Simulator's rabbits and foxes lists. There is a risk that they could be inconsistent with each other. Check that you thoroughly understand how the Field and the animal lists are kept consistent between the simulateOneStep method in Simulator, hunt in Fox, and run in Rabbit.

Exercise 12.26 Do you think it would be better for Simulator not to keep separate lists of foxes and rabbits, but to generate these lists again from the contents of the field at the beginning of each simulation step? Discuss this.
**Exercise 12.27** Write a test to ensure that, at the end of a simulation step, there is no animal (dead or alive) in the field that is not in one of the lists and vice versa. Should there be any dead animals in any of those places at that stage?

### 12.2.6 Taking steps to improve the simulation

Now that we have examined how the simulation operates, we are in a position to make improvements to its internal design and implementation. Making progressive improvements through the introduction of new programming features will be the focus of subsequent sections. There are several points at which we could start, but one of the most obvious weaknesses is that no attempt has been made to exploit the advantages of inheritance in the implementation of the Fox and Rabbit classes, which share a lot of common elements. In order to do this, we shall introduce the concept of an abstract class.

**Exercise 12.28** Identify the similarities and differences between the Fox and Rabbit classes. Make separate lists of the fields, methods, and constructors, and distinguish between the class variables (static fields) and instance variables.

**Exercise 12.29** Candidate methods for placement in a superclass are those that are identical in all subclasses. Which methods are truly identical in the Fox and Rabbit classes? In reaching a conclusion, you might like to consider the effect of substituting the values of class variables into the bodies of the methods that use them.

**Exercise 12.30** In the current version of the simulation, the values of all similarly named class variables are different. If the two values of a particular class variable (BREEDING\_AGE, say) were identical, would it make any difference to your assessment of which methods are truly identical?

### 12.3 Abstract classes

Chapter 10 introduced concepts such as inheritance and polymorphism that we ought to be able to exploit in the simulation application. For instance, the Fox and Rabbit classes share many similar characteristics that suggest they should be subclasses of a common superclass, such as Animal. In this section, we will start to make such changes in order to improve the design and implementation of the simulation as a whole. As with the project in Chapter 10, using a common superclass should avoid code duplication in the subclasses and simplify the code in the client class (here, Simulator). It is important to note that we are undertaking a process of refactoring and that these changes should not change the essential characteristics of the simulation as seen from a user’s viewpoint.
12.3.1 The Animal superclass

For the first set of changes, we will move the identical elements of Fox and Rabbit to an Animal superclass. The project foxes-and-rabbits-v1 provides a copy of the base version of the simulation for you to follow through the changes we make.

- Both Fox and Rabbit define age, alive, field, and location attributes. However, at this point we will only move alive, location, and field to the Animal superclass, and come back to discuss the age field later. As is our normal practice with instance fields, we will keep all of these private in the superclass. The initial values are set in the constructor of Animal, with alive set to true, and field and location passed via super calls from the constructors of Fox and Rabbit.

- These fields will need accessors and mutators, so we can move the existing getLocation, setLocation, isAlive, and setDead from Fox and Rabbit. We will also need to add a getField method in Animal so that direct access to field from the subclass methods run, hunt, giveBirth, and findFood can be replaced.

- In moving these methods, we have to think about the most appropriate visibility for them. For instance, setLocation is private in both Fox and Rabbit, but cannot be kept private in Animal because Fox and Rabbit would not be able to call it. So we should raise it to protected visibility, to indicate that it is for subclasses to call.

- In a similar vein, notice that setDead was public in Rabbit, but private in Fox. Should it therefore be public in Animal? It was public in Rabbit because a fox needs to be able to call a rabbit’s setDead method when it eats its prey. Now that they are sibling classes of a shared superclass, a more appropriate visibility is protected, again indicating that this is a method that is not a part of an animal’s general interface—at least at this stage of the project’s development.

Making these changes is a first step toward eliminating code duplication through the use of inheritance, in much the same way as we did in Chapter 10.

Exercise 12.31 What sort of regression-testing strategy could you establish before undertaking the process of refactoring on the simulation? Is this something you could conveniently automate?

Exercise 12.32 The Randomizer class provides us with a way to control whether the “random” elements of the simulation are repeatable or not. If its useShared field is set to true, then a single Random object is shared between all of the simulation objects. In addition, its reset method resets the starting point for the shared Random object. Use these features as you work on the following exercise, to check that you do not change anything fundamental about the overall simulation as you introduce an Animal class.

Create the Animal superclass in your version of the project. Make the changes discussed above. Ensure that the simulation works in a similar manner as before. You should be able to check this by having the old and new versions of the project open side by side, for instance, and making identical calls on Simulator objects in both, expecting identical outcomes.
Exercise 12.33 How has using inheritance improved the project so far? Discuss this.

12.3.2 Abstract methods

So far, use of the Animal superclass has helped to avoid a lot of the code duplication in the Rabbit and Fox classes, and has potentially made it easier to add new animal types in the future. However, as we saw in Chapter 10, intelligent use of inheritance should also simplify the client class—in this case, Simulator. We shall investigate this now.

In the Simulator class, we have used separate typed lists of foxes and rabbits and per-list iteration code to implement each simulation step. The relevant code is shown in Code 12.4. Now that we have the Animal class, we can improve this. Because all objects in our animal collections are a subtype of Animal, we can merge them into a single collection and hence iterate just once using the Animal type. However, one problem with this is evident from the single-list solution in Code 12.5. Although we know that each item in the list is an Animal, we still have to work out which type of animal it is in order to call the correct action method for its type—run or hunt. We determine the type using the instanceof operator.

```
for(Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {
    Animal animal = it.next();
    if(animal instanceof Rabbit) {
        Rabbit rabbit = (Rabbit) animal;
        rabbit.run(newAnimals);
    }
    else if(animal instanceof Fox) {
        Fox fox = (Fox) animal;
        fox.hunt(newAnimals);
    }
    else {
        System.out.println("found unknown animal");
    }
    // Remove dead animals from the simulation.
    if(!animal.isAlive()) {
        it.remove();
    }
}
```

The fact that in Code 12.5 each type of animal must be tested for and cast separately, and that special code exists for each animal class, is a good sign that we have not taken full advantage of what inheritance has to offer. A better solution is placing a method in the superclass (Animal), letting an animal act, and then overriding it in each subclass so that we have a polymorphic method call to let each animal act appropriately, without the need to test for the specific animal types. This is a standard refactoring technique in situations like this, where we have subtype-specific behavior invoked from a context that only deals with the supertype.
Let us assume that we create such a method, called `act`, and investigate the resulting source code. Code 12.6 shows the code implementing this solution.

```java
// Let all animals act.
for (Iterator<Animal> it = animals.iterator(); it.hasNext(); ) {
    Animal animal = it.next();
    animal.act(newAnimals);
    // Remove dead animals from the simulation.
    if (!animal.isAlive()) {
        it.remove();
    }
}
```

Several observations are important at this point:

- The variable we are using for each collection element (`animal`) is of type `Animal`. This is legal, because all objects in the collection are foxes or rabbits and are all subtypes of `Animal`.

- We assume that the specific action methods (`run` for `Rabbit`, `hunt` for `Fox`) have been renamed `act`. Instead of telling each animal exactly what to do, we are just telling it to “act,” and we leave it up to the animal itself to decide what exactly it wants to do. This reduces coupling between `Simulator` and the individual animal subclasses.

- Because the dynamic type of the variable determines which method is actually executed (as discussed in Chapter 11), the fox’s action method will be executed for foxes, and the rabbit’s method for rabbits.

- Because type checking is done using the static type, this code will compile only if class `Animal` has an `act` method with the right header.

The last of these points is the only remaining problem. Because we are using the statement

```java
animal.act(newAnimals);
```

and the variable `animal` is of type `Animal`, this will compile only if `Animal` defines an `act` method—as we saw in Chapter 11. However, the situation here is rather different from the situation we encountered with the `display` method in Chapter 11. There, the superclass version of `display` had a useful job to do: print the fields defined in the superclass. Here, although each particular animal has a specific set of actions to perform, we cannot describe in any detail the actions for animals in general. The particular actions depend on the specific subtype.

Our problem is to decide how we should define `Animal`’s `act` method.

The problem is a reflection of the fact that no instance of class `Animal` will ever exist. There is no object in our simulation (or in nature) that is just an animal and not also an instance of a more specific subclass. These kinds of classes, which are not intended for creating objects but serve only as superclasses, are known as abstract classes. For animals, for example, we can
say that each animal can act, but we cannot describe exactly how it acts without referring to a more specific subclass. This is typical for abstract classes, and it is reflected in Java constructs.

For the **Animal** class, we wish to state that each animal has an **act** method, but we cannot give a reasonable implementation in class **Animal**. The solution in Java is to declare the method **abstract**. Here is an example of an abstract **act** method:

```java
abstract public void act(List<Animal> newAnimals);
```

An abstract method is characterized by two details:

1. It is prefixed with the keyword **abstract**.
2. It does not have a method body. Instead, its header is terminated with a semicolon.

Because the method has no body, it can never be executed. But we have already established that we do not want to execute an **Animal**’s **act** method, so that is not a problem.

Before we investigate in detail the effects of using an abstract method, we shall introduce more formally the concept of an abstract class.

### 12.3.3 Abstract classes

It is not only methods that can be declared abstract; classes can be declared abstract as well. Code 12.7 shows an example of class **Animal** as an abstract class. Classes are declared abstract by inserting the keyword **abstract** into the class header.

```java
public abstract class Animal
{

  Fields omitted.

  /**
   * Make this animal act - that is: make it do whatever it wants/needs
   * to do.
   *
   * @param newAnimals A list to receive newly born animals.
   */
  abstract public void act(List<Animal> newAnimals);

  Other methods omitted.
}
```

Classes that are not abstract (all classes we have seen previously) are called **concrete classes**. Declaring a class abstract serves several purposes:

- No instances can be created of abstract classes. Trying to use the **new** keyword with an abstract class is an error and will not be permitted by the compiler. This is mirrored in BlueJ: right-clicking on an abstract class in the class diagram will not list any constructors in the pop-up menu. This serves our intention discussed above: we stated that we
did not want instances of class Animal created directly—this class serves only as a superclass. Declaring the class abstract enforces this restriction.

- Only abstract classes can have abstract methods. This ensures that all methods in concrete classes can always be executed. If we allowed an abstract method in a concrete class, we would be able to create an instance of a class that lacked an implementation for a method.

- Abstract classes with abstract methods force subclasses to override and implement those methods declared abstract. If a subclass does not provide an implementation for an inherited abstract method, it is itself abstract, and no instances may be created. For a subclass to be concrete, it must provide implementations for all inherited abstract methods.

Now we can start to see the purpose of abstract methods. Although they do not provide an implementation, they nonetheless ensure that all concrete subclasses have an implementation of this method. In other words, even though class Animal does not implement the act method, it ensures that all existing animals have an implemented act method. This is done by ensuring that

- no instance of class Animal can be created directly, and

- all concrete subclasses must implement the act method.

Although we cannot create an instance of an abstract class directly, we can otherwise use an abstract class as a type in the usual ways. For instance, the normal rules of polymorphism allow us to handle foxes and rabbits as instances of the Animal type. So those parts of the simulation that do not need to know whether they are dealing with a specific subclass can use the superclass type instead.

**Exercise 12.34** Although the body of the loop in Code 12.6 no longer deals with the Fox and Rabbit types, it still deals with the Animal type. Why is it not possible for it to treat each object in the collection simply using the Object type?

**Exercise 12.35** Is it necessary for a class with one or more abstract methods to be defined as abstract? If you are not sure, experiment with the source of the Animal class in the foxes-and-rabbits-v2 project.

**Exercise 12.36** Is it possible for a class that has no abstract methods to be defined as abstract? If you are not sure, change act to be a concrete method in the Animal class by giving it a method body with no statements.

**Exercise 12.37** Could it ever make sense to define a class as abstract if it has no abstract methods? Discuss this.

**Exercise 12.38** Which classes in the java.util package are abstract? Some of them have Abstract in the class name, but is there any other way to tell from the documentation? Which concrete classes extend them?
Exercise 12.39 Can you tell from the API documentation for an abstract class which (if any) of its methods are abstract? Do you need to know which methods are abstract?

Exercise 12.40 Review the overriding rules for methods and fields discussed in Chapter 11. Why are they particularly significant in our attempts to introduce inheritance into this application?

Exercise 12.41 The changes made in this section have removed the dependencies (couplings) of the simulateOneStep method on the Fox and Rabbit classes. The Simulator class, however, is still coupled to Fox and Rabbit, because these classes are referenced in the populate method. There is no way to avoid this; when we create animal instances, we have to specify exactly what kind of animal to create.

This could be improved by splitting the Simulator into two classes: one class, Simulator, that runs the simulation and is completely decoupled from the concrete animal classes, and another class, PopulationGenerator (created and called by the simulator), that creates the population. Only this class is coupled to the concrete animal classes, making it easier for a maintenance programmer to find places where change is necessary when the application is extended. Try implementing this refactoring step. The PopulationGenerator class should also define the colors for each type of animal.

The project foxes-and-rabbits-v2 provides an implementation of our simulation with the improvements discussed here. It is important to note that the change in Simulator to processing all the animals in a single list, rather than in separate lists, means that the simulation results in version 2 will not be identical to those in version 1.

In the book projects, you will find a third version of this project: foxes-and-rabbits-graph. This project is identical to foxes-and-rabbits-v2 in its model (i.e., the animal/fox/rabbit/simulator implementations), but it adds a second view to the project: a graph showing population numbers over time. We will discuss some aspects of its implementation a little later in this chapter; for now, just experiment with this project.

Exercise 12.42 Open and run the foxes-and-rabbits-graph project. Pay attention to the Graph View output. Explain, in writing, the meaning of the graph you see, and try to explain why it looks the way it looks. Is there a relationship between the two curves?

Exercise 12.43 Repeat some of your experiments with different sizes of fields (especially smaller fields). Does the Graph View give you any new insights or help you understand or explain what you see?
If you have done all the exercises in this chapter so far, then your version of the project will be the same as `foxes-and-rabbits-v2` and similar to `foxes-and-rabbits-graph`, except for the graph display. You can continue the exercises from here on with either version of these projects.

## 12.4 More abstract methods

When we created the `Animal` superclass in Section 12.3.1, we did this by identifying common elements of the subclasses, but we chose not to move the `age` field and the methods associated with it. This might be overly conservative. We could have easily moved the `age` field to `Animal` and provided for it there an accessor and a mutator that were called by subclass methods, such as `incrementAge`. Why didn’t we move `incrementAge` and `canBreed` into `Animal`, then? The reason for not moving these is that, although several of the remaining method bodies in Fox and Rabbit contain textually identical statements, their use of class variables with different values means that they cannot be moved directly to the superclass. In the case of `canBreed`, the problem is the `BREEDING_AGE` variable, while `breed` depends on `BREEDING_PROBABILITY` and `MAX_LITTER_SIZE`. If `canBreed` is moved to `Animal`, for instance, then the compiler will need to have access to a value for the subtype-specific breeding age in class `Animal`. It is tempting to define a `BREEDING_AGE` field in the `Animal` class and assume that its value will be overridden by similarly named fields in the subclasses. However, fields are handled differently from methods in Java: they cannot be overridden by subclass versions.\(^3\) This means that a `canBreed` method in `Animal` would use a meaningless value defined in that class, rather than one that is specific to a particular subclass.

The fact that the field’s value would be meaningless gives us a clue as to how we can get around this problem and, as a result, move more of the similar methods from the subclasses to the superclass.

Remember that we defined `act` as abstract in `Animal` because having a body for the method would be meaningless. If we access the breeding age with a method rather than a field, we can get around the problems associated with the age-dependent properties. This approach is shown in Code 12.8.

```java
/**
 * An animal can breed if it has reached the breeding age.
 * @return true if the animal can breed
 */
public boolean canBreed()
{
    return age >= getBreedingAge();
}

/**
 * Return the breeding age of this animal.
 * @return The breeding age of this animal.
 */
abstract protected int getBreedingAge();

\(^3\) This rule applies regardless of whether a field is static or not.
The `canBreed` method has been moved to `Animal` and rewritten to use the value returned from a method call rather than the value of a class variable. For this to work, a method `getBreedingAge` must be defined in class `Animal`. Because we cannot specify a breeding age for animals in general, we can again use an `abstract` method in the `Animal` class and concrete redefinitions in the subclasses. Both `Fox` and `Rabbit` will define their own versions of `getBreedingAge` to return their particular values of `BREEDING_AGE`:

```java
/**
 * @return The age at which a rabbit starts to breed.
 */

public int getBreedingAge()
{
    return BREEDING_AGE;
}
```

So even though the call to `getBreedingAge` originates in the code of the superclass, the method called is defined in the subclass. This may seem mysterious at first but it is based on the same principles we described in Chapter 11 in using the dynamic type of an object to determine which version of a method is called at runtime. The technique illustrated here makes it possible for each instance to use the value appropriate to its subclass type. Using the same approach, we can move the remaining methods, `incrementAge` and `breed`, to the superclass.

**Exercise 12.44** Using your latest version of the project (or the `foxes-and-rabbits-v2` project in case you have not done all the exercises), record the number of foxes and rabbits over a small number of steps, to prepare for regression testing of the changes to follow.

**Exercise 12.45** Move the `age` field from `Fox` and `Rabbit` to `Animal`. Initialize it to zero in the constructor. Provide accessor and mutator methods for it and use these in `Fox` and `Rabbit` rather than in direct accesses to the field. Make sure the program compiles and runs as before.

**Exercise 12.46** Move the `canBreed` method from `Fox` and `Rabbit` to `Animal`, and rewrite it as shown in Code 12.8. Provide appropriate versions of `getBreedingAge` in `Fox` and `Rabbit` that return the distinctive breeding age values.

**Exercise 12.47** Move the `incrementAge` method from `Fox` and `Rabbit` to `Animal` by providing an abstract `getMaxAge` method in `Animal` and concrete versions in `Fox` and `Rabbit`.

**Exercise 12.48** Can the `breed` method be moved to `Animal`? If so, make this change.

**Exercise 12.49** In light of all the changes you have made to these three classes, reconsider the visibility of each method and make any changes you feel are appropriate.
Exercise 12.50 Was it possible to make these changes without having any impact on any other classes in the project? If so, what does this suggest about the degrees of encapsulation and coupling that were present in the original version?

Exercise 12.51 Challenge exercise Define a completely new type of animal for the simulation, as a subclass of Animal. You will need to decide what sort of impact its existence will have on the existing animal types. For instance, your animal might compete with foxes as a predator on the rabbit population, or your animal might prey on foxes but not on rabbits. You will probably find that you need to experiment quite a lot with the configuration settings you use for it. You will need to modify the populate method to have some of your animals created at the start of a simulation.

You should also define a new color for your new animal class. You can find a list of predefined color names on the API page documenting the Color class in the java.awt package.

Exercise 12.52 Challenge exercise The text of the giveBirth methods in Fox and Rabbit is very similar. The only difference is that one creates new Fox objects and the other creates new Rabbit objects. Is it possible to use the technique illustrated with canBreed to move the common code into a shared giveBirth method in Animal? If you think it is, try it out. Hint: The rules on polymorphic substitution apply to values returned from methods as well as in assignment and parameter passing.

12.5 Multiple inheritance

12.5.1 An Actor class

In this section, we discuss some possible future extensions and some programming constructs to support these.

The first obvious extension for our simulation is the addition of new animals. If you have attempted Exercise 12.51 then you will have touched on this already. We should, however, generalize this a bit: maybe not all participants in the simulation will be animals. Our current structure assumes that all acting participants in the simulation are animals and that they inherit from the Animal superclass. One enhancement that we might like to make is the introduction of human predators to the simulation, as either hunters or trappers. They do not neatly fit the existing assumption of purely animal-based actors. We might also extend the simulation to include plants eaten by the rabbits, or even some aspects of the weather. The plants as food would influence the population of rabbits (in effect, rabbits become predators of the plants), and the growth of the plants might be influenced by the weather. All these new components would act in the simulation, but they are clearly not animals, so it would be inappropriate to have them as subclasses of Animal.
As we consider the potential for introducing further actors into the simulation, it is worth revealing why we chose to store details of the animals in both a Field object and an Animal list. Visiting each animal in the list is what constitutes a single simulation step. Placing all participants in a single list keeps the basic simulation step simple. However, this clearly duplicates information, which risks creating inconsistency. One reason for this design decision is that it allows us to consider participants in the simulation that are not actually within the field—a representation for the weather might be one example of this.

To deal with more general actors, it seems like a good idea to introduce an Actor superclass. The Actor class would serve as a superclass to all kinds of simulation participants, independent of what they are. Figure 12.3 shows a class diagram for this part of the simulation. The Actor and Animal classes are abstract, while Rabbit, Fox, and Hunter are concrete classes.

The Actor class would include the common part of all actors. One thing all possible actors have in common is that they perform some kind of action. We will also need to know whether an actor is still active or not. So the only definitions in class Actor are those of abstract act and isActive methods:

```java
// all comments omitted
public abstract class Actor
{
    abstract public void act(List<Actor> newActors);
    abstract public boolean isActive();
}
```

This should be enough to rewrite the actor loop in the Simulator (Code 12.6), using class Actor instead of class Animal. (Either the isAlive method could be renamed to isActive, or a separate isActive method in Animal could simply call the existing isAlive method.)

---

**Figure 12.3**
Simulation structure with Actor
Exercise 12.53 Introduce the Actor class into your simulation. Rewrite the simulateOneStep method in Simulator to use Actor instead of Animal. You can do this even if you have not introduced any new participant types. Does the Simulator class compile? Or is there something else that is needed in the Actor class?

This new structure is more flexible because it allows easier addition of non-animal actors. In fact, we could even rewrite the statistics-gathering class, FieldStats, as an Actor—it too acts once every step. Its action would be to update its current count of animals.

12.5.2 Flexibility through abstraction

By moving towards the notion of the simulation being responsible for managing actor objects, we have succeeded in abstracting quite a long way away from our original very specific scenario of foxes and rabbits in a rectangular field. This process of abstraction has brought with it an increased flexibility that may allow us to widen even further the scope of what we might do with a general simulation framework. If we think through the requirements of other similar simulation scenarios, then we might come up with ideas for additional features that we could introduce.

For instance, it might be useful to simulate other predator–prey scenarios such as a marine simulation involving fish and sharks, or fish and fishing fleets. If the marine simulation were to involve modeling food supplies for the fish, then we would probably not want to visualize plankton populations—either because the numbers are too vast, or because their size is too small. Environmental simulations might involve modeling the weather, which, while it is clearly an actor, also might not require visualization in the field cells.

In the next section, we shall investigate the separation of visualization from acting, as a further extension to our simulation framework.

12.5.3 Selective drawing

One way to implement the separation of visualization from acting is to change the way it is performed in the simulation. Instead of iterating over the whole field every time and drawing actors in every position, we could iterate over a separate collection of drawable actors. The code in the simulator class would look similar to this:

```java
// Let all actors act.
for (Actor actor : actors) {
    actor.act(...);
}

// Draw all drawables.
for (Drawable item : drawables) {
    item.draw(...);
}
```
All of the actors would be in the actors collection, and those actors we want to show on screen would also be in the drawables collection. For this to work, we need another superclass called Drawable, which declares an abstract draw method. Drawable actors must then inherit from both Actor and Drawable. (Figure 12.4 shows an example where we assume that we have ants, which act but are too numerous to visualize.)

12.5.4 Drawable actors: multiple inheritance

The scenario presented here uses a structure known as multiple inheritance. Multiple inheritance exists in cases where one class has more than one immediate superclass. The subclass then has all the features of both superclasses and those defined in the subclass itself.

Multiple inheritance is quite easy to understand in principle but can lead to significant complications in the implementation of a programming language. Different object-oriented languages vary in their treatment of multiple inheritance: some languages allow the inheritance of multiple superclasses; others do not. Java lies somewhere in the middle. It does not allow multiple inheritance of classes, but provides another construct, called an “interface,” that allows a limited form of multiple inheritance. Interfaces are discussed in the next section.

12.6 Interfaces

Up to this point in the book, we have used the term “interface” in an informal sense, to represent that part of a class that couples it to other classes. Java captures this concept more formally by allowing interface types to be defined.

---

4 Don't confuse this case with the regular situation where a single class might have several superclasses in its inheritance hierarchy, such as Fox, Animal, Actor, and Object. This is not what is meant by multiple inheritance.
At first glance, interfaces are similar to classes. In their most common form they are closest to abstract classes in which all the methods are abstract. We can summarize the most significant features of interfaces as follows:

- The keyword `interface` is used instead of `class` in the header of the declaration.
- Interfaces do not contain any constructors.
- Interfaces do not contain any instance fields.
- Only fields that are constant class fields (`static` and `final`) with public visibility are allowed in an interface. The `public`, `static`, and `final` keywords may be omitted, therefore; they are assumed automatically.
- Abstract methods do not have to include the keyword `abstract` in their header.

Prior to Java 8, all methods in an interface had to be abstract, but the following non-abstract method types are also now available in interfaces:

- Methods marked with the `default` keyword have a method body.
- Methods marked with the `static` keyword have a method body.

All methods in an interface—whether abstract, concrete or static—have public visibility, so the `public` keyword may be omitted in their definition.

### 12.6.1 An Actor interface

Code 12.9 shows `Actor` defined as an interface type.

```java
/**
 * The interface to be extended by any class wishing
 * to participate in the simulation.
 */

public interface Actor
{
    /**
     * Perform the actor's regular behavior.
     * @param newActors A list for receiving newly created actors.
     */
    void act(List<Actor> newActors);

    /**
     * Is the actor still active?
     * @return true if still active, false if not.
     */
    boolean isActive();
}
```

A class can inherit from an interface in a similar way to that for inheriting from a class. However, Java uses a different keyword—`implements`—for inheriting interfaces.
A class is said to implement an interface if it includes an implements clause in its class header. For instance:

```java
public class Fox extends Animal implements Drawable
{
  Body of class omitted.
}
```

As in this case, if a class both extends a class and implements an interface, then the extends clause must be written first in the class header.

Two of our abstract classes in the example above, Actor and Drawable, are good candidates for being written as interfaces. Both of them contain only the definition of methods, without method implementations. Thus, they already fit the most abstract definition of an interface perfectly: they contain no instance fields, no constructors, and no method bodies.

The class Animal is a different case. It is a real abstract class that provides a partial implementation with instance fields, a constructor and many methods with bodies. It has only a single abstract method in its original version. Given all of these characteristics, it must remain as a class rather than becoming an interface.

**Exercise 12.54** Redefine as an interface the abstract class Actor in your project. Does the simulation still compile? Does it run? Make any changes necessary to make it runnable again.

**Exercise 12.55** Are the fields in the following interface class fields or instance fields?

```java
public interface Quiz
{
  int CORRECT = 1;
  int INCORRECT = 0;
  ...
}
```

What visibility do they have?

**Exercise 12.56** What are the errors in the following interface?

```java
public interface Monitor
{
  private static final int THRESHOLD = 50;
  private int value;
  public Monitor (int initial);
  void update(int reading);
  int getThreshold()
  {
    return THRESHOLD;
  }
  ...
}
```
12.6.2 Default methods in interfaces

A method marked as `default` in an interface will have a method body, which is inherited by all implementing classes. The addition of default methods to interfaces in Java 8 muddied the waters in the distinction between abstract classes and interfaces, since it is no longer true that interfaces never contain method bodies. However, it is important to exercise caution when considering defining a default method in an interface. It should be borne in mind that default methods were added to the language primarily to support the addition of new methods to interfaces in the API that existed before Java 8. Default methods made it possible to change existing interfaces without breaking the many classes that already implemented the older versions of those interfaces.

From the other limitations of interfaces—no constructors and no instance fields—it should be clear that the functionality possible in a default method is strictly limited, since there is no state that can be examined or manipulated directly by them. In general, therefore, when writing our own interfaces we will tend to limit ourselves to purely abstract methods. Furthermore, when discussing features of interfaces in this chapter we will often ignore the case of non-abstract methods for the sake of simplicity.

12.6.3 Multiple inheritance of interfaces

As mentioned above, Java allows any class to extend at most one other class. However, it allows a class to implement any number of interfaces (in addition to possibly extending one class). Thus, if we define both `Actor` and `Drawable` as interfaces instead of abstract classes, we can define class `Hunter` (Figure 12.4) to implement both of them:

```java
class Hunter implements Actor, Drawable
{
    // Body of class omitted.
}
```

The class `Hunter` inherits the methods of all interfaces (`act` and `draw`, in this case) as abstract methods. It must, then, provide method definitions for both of them by overriding the methods, or the class itself must be declared abstract.

The `Animal` class shows an example where a class does not implement an inherited interface method. `Animal`, in our new structure in Figure 12.4, inherits the abstract method `act` from `Actor`. It does not provide a method body for this method, which makes `Animal` itself abstract (it must include the `abstract` keyword in the class header). The `Animal` subclasses then implement the `act` method and become concrete classes.

The presence of default methods in interfaces can lead to complications in the implementation of multiple interfaces by a class. Where a class implements multiple interfaces and two or more of the interfaces have a default method with the same signature then the implementing class must override that method—even if the alternative versions of the method are identical. The reason for this is that, in the general case, it needs to be clear exactly which of the alternative implementations should be inherited by the class. The class may override either by calling the preferred version in the overriding method, or by defining a completely different implementation. The header of the overriding method will not contain the `default` keyword, since that is only used in interfaces.
There is a new syntax involving the keyword `super` for calling a default method from one of the interfaces in an overriding method. For instance, suppose the `Actor` and `Drawable` interfaces both define a default method called, `reset`, that has a `void` return type and takes no parameters. If a class implementing both interfaces overrides this method by calling both default versions then the overriding method might be defined as follows:

```java
public void reset()
{
    Actor.super.reset();
    Drawable.super.reset();
}
```

In other words, an implementing class that overrides an inherited default method can call that default method using the syntax:

```java
InterfaceName.super.methodName(...)
```

**Exercise 12.57** *Challenge exercise* Add a non-animal actor to the simulation. For instance, you could introduce a `Hunter` class with the following properties. Hunters have no maximum age and neither feed nor breed. At each step of the simulation, a hunter moves to a random location anywhere in the field and fires a fixed number of shots into random target locations around the field. Any animal in one of the target locations is killed.

Place just a small number of hunters in the field at the start of the simulation. Do the hunters remain in the simulation throughout, or do they ever disappear? If they do disappear, why might that be, and does that represent realistic behavior?

What other classes required changing as a result of introducing hunters? Is there a need to introduce further decoupling to the classes?

### 12.6.4 Interfaces as types

When a class implements an interface, it often does not inherit any implementation from it. The question, then, is: What do we actually gain by implementing interfaces?

When we introduced inheritance in Chapter 10, we emphasized two great benefits of inheritance:

1. The subclass inherits the code (method implementations and fields) from the superclass. This allows reuse of existing code and avoids code duplication.

2. The subclass becomes a subtype of the superclass. This allows polymorphic variables and method calls. In other words, it allows different special cases of objects (instances of subclasses) to be treated uniformly (as instances of the supertype).

Interfaces are not primarily used for the first benefit but for the second. An interface defines a type just as a class does. This means that variables can be declared to be of interface types, even though no objects of that type can exist (only subtypes).
In our example, even though `Actor` is now an interface, we can still declare an `Actor` variable in the `Simulator` class. The simulation loop still works unchanged.

Interfaces can have no direct instances, but they serve as supertypes for instances of other classes.

### 12.6.5 Interfaces as specifications

In this chapter, we have introduced interfaces as a means to implement multiple inheritance in Java. This is one important use of interfaces, but there are others.

The most important characteristic of interfaces is that they almost completely separate the definition of the functionality (the class’s “interface” in the wider sense of the word) from its implementation. A good example of how this can be used in practice can be found in the Java collection hierarchy.

The collection hierarchy defines (among other types) the interface `List` and the classes `ArrayList` and `LinkedList` (Figure 12.5). The `List` interface specifies the full functionality of a list, without constraining its underlying structural implementation. The subclasses (`LinkedList` and `ArrayList`) provide two completely different structural implementations of the same interface. This is interesting, because the two implementations differ greatly in the efficiency of some of their functions. Random access to elements in the middle of the list, for example, is much faster with the `ArrayList`. Inserting or deleting elements, however, can be much faster in the `LinkedList`.

Which implementation is better in any given application can be hard to judge in advance. It depends a lot on the relative frequency with which certain operations are performed and on some other factors. In practice, the best way to find out is often to try it out: implement the application with both alternatives and measure the performance.

The existence of the `List` interface makes it very easy to do this. If, instead of using `ArrayList` or `LinkedList` as variable and parameter types, we always use `List`, our application will work independently of the specific type of list we are currently using. Only when we create a new list do we really have to use the name of the specific implementation. We would, for instance, write

```java
List<Type> myList = new ArrayList<>();
```

**Figure 12.5**
The `List` interface and its subclasses
Note that the polymorphic variable's type is just `List` of `Type`. This way, we can change the whole application to use a linked list by just changing `ArrayList` to `LinkedList` in a single location when the list is being created.

### 12.6.6 Library support through abstract classes and interfaces

In Chapter 6, we pointed out the importance of paying attention to the names of the collection classes: `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, etc. Now that we have been introduced to abstract classes and interfaces, we can see why these particular names have been chosen. The `java.util` package defines several important collection abstractions in the form of interfaces, such as `List`, `Map`, and `Set`. The concrete class names have been chosen to communicate information about both what kind of interface they conform to and some of the underlying implementation detail. This information is very useful when it comes to making informed decisions about the right concrete class to use in a particular setting. However, by using the highest-level abstract type (be it abstract class or interface) for our variables, wherever possible, our code will remain flexible in light of future library changes—such as the addition of a new `Map` or `Set` implementation, for instance.

In Chapter 13, where we introduce the Java GUI libraries, we will be making enormous use of abstract classes and interfaces as we see how to create quite sophisticated functionality with very little additional code.

**Exercise 12.58** Which methods do `ArrayList` and `LinkedList` have that are not defined in the `List` interface? Why do you think that these methods are not included in `List`?

**Exercise 12.59** Write a class that can make comparisons between the efficiency of the common methods from the `List` interface in the `ArrayList` and `LinkedList` classes such as `add`, `get`, and `remove`. Use the polymorphic-variable technique described above to write the class so that it only knows it is performing its tests on objects of the interface type `List` rather than on the concrete types `ArrayList` and `LinkedList`. Use large lists of objects for the tests, to make the results significant. You can use the `currentTimeMillis` method of the `System` class for getting hold of the start and finish time of your test methods.

**Exercise 12.60** Read the API description for the `sort` methods of the `Collections` class in the `java.util` package. Which interfaces are mentioned in the descriptions? Which methods in the `java.util.List` interface have `default` implementations?

**Exercise 12.61** Challenge exercise: Investigate the `Comparable` interface. This is a parameterized interface. Define a simple class that implements `Comparable`. Create a collection containing objects of this class and sort the collection. Hint: The `LogEntry` class of the `weblog-analyzer` project in Chapter 7 implements this interface.
12.6.7 Functional interfaces and lambdas (advanced)

Java 8 introduced a special classification for interfaces that contain just a single abstract method (regardless of the number of default and/or static methods they contain). Such an interface is called a functional interface. The annotation @FunctionalInterface may be included with the declaration to allow the compiler to check that the interface conforms to the rules for functional interfaces.

There is a special relationship between functional interfaces and lambda expressions. Anywhere that an object of a functional interface type is required, a lambda expression may be used instead. In the following chapter, we shall see extensive use of this feature when implementing graphical user interfaces.

This link between lambdas and functional interfaces is important. In particular, it gives us a convenient means to associate a lambda expression with a type, for example to declare a variable that can hold a lambda. The java.util.function package defines a large number of interfaces that provide convenience names for the most commonly occurring types of lambda expression. In general, the interface names indicate the return type and the parameter types of their single method, and hence a lambda of that type.

For example:

- Consumer interfaces relate to lambdas with a void return type. For instance, DoubleConsumer takes a single double parameter and returns no result.
- BinaryOperator interfaces take two parameters and return a result of the same type. For instance, IntBinaryOperator takes two int parameters and returns an int result.
- Supplier interfaces return a result of the indicated type. For instance, LongSupplier.
- Predicate interfaces return a boolean result. For instance, IntPredicate.

All of these interfaces extend the Function interface, whose single abstract method is called apply. One potentially useful functional interface type, defined in the java.lang package, is Runnable. This fills a gap in the list of Consumer interfaces in that it takes no parameters and has a void return type. However, its single abstract method is called run.

Functional interface types allow lambdas to be assigned to variables or passed as actual parameters. For example, suppose we have pairs of name and alias strings that we wish to format in a particular way, such as "Michelangelo Merisi (AKA Caravaggio)". A lambda taking two String parameters and returning a String result is compatible with the BinaryOperator interface, and we might give a type and name to a lambda as follows:

```java
BinaryOperator<String> aka =
    (name, alias) -> return name + " (AKA " + alias + ")";
```

This lambda would be used by calling its apply method with appropriate parameters, for instance:

```java
System.out.println(aka.apply("Michelangelo Merisi", "Caravaggio"));
```
12.7 A further example of interfaces

In the previous section, we have discussed how interfaces can be used to separate the specification of a component from its implementation so that different implementations can be "plugged in," thus making it easy to replace components of a system. This is often done to separate parts of a system that are logically only loosely coupled.

We have seen an example of this (without discussing it explicitly) at the end of Section 12.3. There, we investigated the foxes-and-rabbits-graph project, which added another view of the populations in the form of a line graph. Looking at the class diagram for this project, we can see that the addition also makes use of a Java interface (Fig 12.6).

The previous versions of the foxes-and-rabbits project contained only one SimulatorView class. This was a concrete class, and it provided the implementation of a grid-based view of the field. As we have seen, the visualization is quite separate from the simulation logic (the field and the actors), and different visualization views are possible.

For this project, SimulatorView was changed from a class to an interface, and the implementation of this view was moved into a class named GridView.

GridView is identical to the previous SimulatorView class. The new SimulatorView interface was constructed by searching through the Simulator class to find all methods that are actually called from outside, then defining an interface that specifies exactly those methods. They are:

```
view.setColor(classObject, color);
view.isViable(field);
view.showStatus(step, field);
view.reset();
```

We can now easily define the complete SimulatorView interface:

```java
import java.awt.Color;
public interface SimulatorView
{
    void setColor(Class<?> animalClass, Color color);
    boolean isViable(Field field);
    void showStatus(int step, Field field);
    void reset();
}
```

Figure 12.6
The SimulatorView interface and implementing classes
The one slightly tricky detail in the definition above is the use of the type `Class<?>` as the first parameter of the `setColor` method. We will explain that in the next section.

The previous `SimulatorView` class, now called `GridView`, is specified to implement the new `SimulatorView` interface:

```java
class GridView extends JFrame implements SimulatorView
{
    ...
}
```

It does not require any additional code, because it already implements the interface’s methods. However, after making these changes, it becomes fairly easy to “plug in” other views for the simulation by providing further implementations of the `SimulatorView` interface. The new class `GraphView`, which produces the line graph, is an example of this.

Once we have more than one view implementation, we can easily replace the current view with another or, as we have in our example, even display two views at the same time. In the `Simulator` class, the concrete subclasses `GridView` and `GraphView` are only mentioned once when each view was constructed. Thereafter, they are stored in a collection holding elements of the `SimulatorView` supertype, and only the interface type is used to communicate with them.

The implementations of the `GridView` and `GraphView` classes are fairly complex, and we do not expect you to fully understand them at this stage. The pattern of providing two implementations for a single interface, however, is important here, and you should make sure that you understand this aspect.

**Exercise 12.62** Review the source code of the `Simulator` class. Find all occurrences of the view classes and interfaces, and trace all variables declared using any of these types. Explain exactly how the views are used in the `Simulator` class.

**Exercise 12.63** Implement a new class `TextView` that implements `SimulatorView`. `TextView` provides a textual view of the simulation. After every simulation step, it prints out one line in the form

```
Foxes: 121 Rabbits: 266
```

Use `TextView` instead of `GridView` for some tests. (Do not delete the `GridView` class. We want to have the ability to change between different views!)

**Exercise 12.64** Can you manage to have all three views active at the same time?
12.8 The Class class

In Chapter 10, we described the paradoxically named Object class. It shouldn’t surprise you, therefore, that there is also a Class class! This is where talking about classes and objects can become very confusing.

We used the Class type in defining the SimulatorView interface in the previous section. The Class class has nothing specifically to do with interfaces; it is a general feature of Java, but we just happen to be meeting it for the first time here. The idea is that each type has a Class object associated with it.

The Object class defines the method getClass to return the Class associated with an object. Another way to get the Class object for a type is to write “.class” after the type name: for instance, Fox.class or int.class—notice that even the primitive types have Class objects associated with them.

The Class class is a generic class—it has a type parameter specifying the specific class subtype we are referencing. For example, the type of String.class is Class<String>. We can use a question mark in place of the type parameter—Class<?>—if we want to declare a variable that can hold all class objects of all types.

Class objects are particularly useful if we want to know whether the type of two objects is the same. We use this feature in the original SimulatorView class to associate each animal type with a color in the field. SimulatorView has the following field to map one to the other:

```java
private Map<Class<?>, Color> colors;
```

When the view is set up, the constructor of Simulator has the following calls to its setColor method:

```java
view.setColor(Rabbit.class, Color.ORANGE);
view.setColor(Fox.class, Color.BLUE);
```

We won’t go into any further detail about Class<?>, but this description should be sufficient to enable you to understand the code shown in the previous section.

12.9 Abstract class or interface?

In some situations, a choice has to be made between whether to use an abstract class or an interface, while in other cases, either abstract classes or interfaces can do the job. Prior to the introduction of default methods in Java 8, the issue appeared to be clearer: if a type needed elements of concrete implementation—such as instance fields, constructors, or method bodies—then an abstract class would have to be used. However, the present availability of default methods in interfaces should not really change the answer to this question in most cases. In general, it is preferable to avoid defining default methods in interfaces except for the purpose of adapting legacy code.
If we have a choice, interfaces are usually preferable. Interfaces are relatively lightweight types that minimize constraints on implementing classes. Furthermore, if we provide a type as an abstract class, then subclasses cannot extend any other classes. Because interfaces allow multiple inheritance, the use of an interface does not create such a restriction. Interfaces cleanly separate the type specification from the implementation, and this creates less coupling. Therefore, using interfaces leads to a more flexible and more extensible structure.

### 12.10 Event-driven simulations

The style of simulation we have used in this chapter has the characteristic of time passing in discrete, equal-length steps. At each time step, each actor in the simulation was asked to act—i.e., take the actions appropriate to its current state. This style of simulation is sometimes called *time-based*, or *synchronous*, simulation. In this particular simulation, most of the actors will have had something to do at each time step: move, breed, and eat. In many simulation scenarios, however, actors spend large numbers of time steps doing nothing—typically, waiting for something to happen that requires some action on their part. Consider the case of a newly born rabbit in our simulation, for instance. It is repeatedly asked whether it is going to breed, even though it takes several time steps before this is possible. Isn’t there a way to avoid asking this unnecessary question until the rabbit is actually ready?

There is also the question of the most appropriate size of the time step. We deliberately left vague the issue of how much real time a time step represents, and the various actions actually require significantly different amounts of time (eating and movement should occur much more frequently than giving birth, for instance). Is there a way to decide on a time-step size that is not so small that most of the time nothing will be happening, or too long that different types of actions are not distinguished clearly enough between time steps?

An alternative approach is to use an *event-based*, or *asynchronous*, simulation style. In this style, the simulation is driven by maintaining a schedule of future events. The most obvious difference between the two styles is that, in an event-based simulation, time passes in uneven amounts. For instance, one event might occur at time \( t \) and the next two events occur at times \( t + 2 \) and time \( t + 8 \), while the following three events might all occur at time \( t + 9 \).

For a fox-and-rabbits simulation, the sort of events we are talking about would be birth, movement, hunting, and death from natural causes. What typically happens is that, as each event occurs, a fresh event is scheduled for some point in the future. For instance, when a birth event occurs, the event marking that animal’s death from old age will be scheduled. All future events are stored in an ordered queue, where the next event to take place is held at the head of the queue. It is important to appreciate that newly scheduled events will not always be placed at the end of the current queue; they will often have to be inserted somewhere before the end, in order to keep the queue in time order. In addition, some future events will be rendered obsolete by events that occur before them—an obvious example is that the natural-death event for a rabbit will not take place if the rabbit is eaten beforehand!
Event-driven simulations lend themselves particularly well to the techniques we have described in this chapter. For instance, the concept of an event is likely to be implemented as an `Event` abstract class containing concrete details of when the event will occur, but only abstract details of what the event involves. Concrete subclasses of `Event` will then supply the specific details for the different event types. Typically, the main simulation loop will not need to be concerned with the concrete event types, but will be able to use polymorphic method calls when an event occurs.

Event-based simulations are often more efficient and are preferable where large systems and large amounts of data are involved, while synchronous simulations are better for producing time-based visualizations (such as animations of the actors) because time flows more evenly.

**Exercise 12.65** Find out some more about how event-driven simulations differ from time-based simulations.

**Exercise 12.66** Look at the `java.util` package to see if there are any classes that might be well suited to storing an event queue in an event-based simulation.

**Exercise 12.67** Challenge exercise Rewrite the foxes-and-rabbits simulation in the event-based style.

### 12.11 Summary of inheritance

In Chapters 10 through 12, we have discussed many different aspects of inheritance techniques. These include code inheritance and subtyping as well as inheriting from interfaces, abstract classes, and concrete classes.

In general, we can distinguish two main purposes of using inheritance: we can use it to inherit code (code inheritance), and we can use it to inherit the type (subtyping). The first is useful for code reuse, the second for polymorphism and specialization.

When we inherit from ("extend") concrete classes, we do both: we inherit the implementation and the type. When we inherit from ("implement") interfaces, we separate the two: we inherit a type but (usually) no implementation. For cases where parts of both are useful, we can inherit from abstract classes; here, we inherit the type and a partial implementation.

When inheriting a complete implementation, we can choose to add or override methods. When no or only partial implementation of a type is inherited, the subclass must provide the implementation before it can be instantiated.

Some other object-oriented languages also provide mechanisms to inherit code without inheriting the type. Java does not provide such a construct.
Summary

In this chapter, we have discussed the fundamental structure of computer simulations. We have then used this example to introduce abstract classes and interfaces as constructs that allow us to create further abstractions and develop more-flexible applications.

Abstract classes are classes that are not intended to have any instances. Their purpose is to serve as superclasses to other classes. Abstract classes may have both abstract methods—methods that have a header but no body—and full method implementations. Concrete subclasses of abstract classes must override abstract methods to provide the missing method implementations.

Another construct for defining types in Java is the interface. Java interfaces are similar to completely abstract classes: they define method headers, but generally provide no implementation. Interfaces define types that can be used for variables.

Interfaces can be used to provide a specification for a class (or part of an application) without stating anything about the concrete implementation.

Java allows multiple inheritance of interfaces (which it calls "implements" relationships) but only single inheritance for classes ("extends" relationships). Multiple inheritance is made more complicated by the presence of conflicting default methods.

Terms introduced in this chapter:

abstract method, abstract class, concrete class, abstract subclass, multiple inheritance, interface (Java construct), implements
Exercise 12.68  Can an abstract class have concrete (non-abstract) methods? Can a concrete class have abstract methods? Can you have an abstract class without abstract methods? Justify your answers.

Exercise 12.69  Look at the code below. You have five types—classes or interfaces—(U, G, B, Z, and X) and a variable of each of these types.

```java
U u;
G g;
B b;
Z z;
X x;
```

The following assignments are all legal (assume that they all compile).

```java
u = z;
x = b;
g = u;
x = u;
```

The following assignments are all illegal (they cause compiler errors).

```java
u = b;
x = g;
b = u;
z = u;
g = x;
```

What can you say about the types and their relationships? (What relationship are they to each other?)

Exercise 12.70  Assume that you want to model people in a university to implement a course management system. There are different people involved: staff members, students, teaching staff, support staff, tutors, technical-support staff, and student technicians. Tutors and student technicians are interesting: tutors are students who have been hired to do some teaching, and student technicians are students who have been hired to help with the technical support.

Draw a type hierarchy (classes and interfaces) to represent this situation. Indicate which types are concrete classes, abstract classes, and interfaces.

Exercise 12.71  Challenge exercise Sometimes class/interface pairs exist in the Java standard library that define exactly the same methods. Often, the interface name ends with Listener and the class name with Adapter. An example is PrintJobListener and PrintJobAdapter. The interface defines some method headers, and the adapter class defines the same methods, each with an empty method body. What might the reason be for having them both?
Exercise 12.72 The collection library has a class named TreeSet, which is an example of a sorted set. Elements in this set are kept in order. Carefully read the description of this class, and then write a class Person that can be inserted into a TreeSet, which will then sort the Person objects by age.

Exercise 12.73 Use the API documentation for the AbstractList class to write a concrete class that maintains an unmodifiable list.