- **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.

Code 10.5 Source code of the **NewsFeed** class (second version)

```
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)

```
    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
        }
    }
}
```

Concept

Subtype As an analog to the class hierarchy, types form a type hierarchy. The type defined by a subclass definition is a subtype of the type of its superclass.

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:

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

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

```
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,

```
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.

Concept

Variables and subtypes Variables may hold objects of their declared type or of any subtype of their 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:

```
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

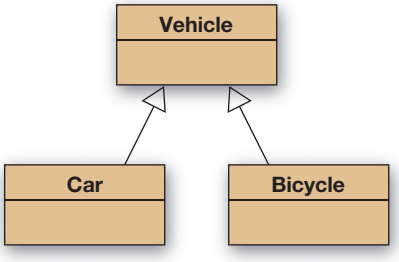

Figure 10.9
An inheritance hierarchy

Concept

Substitution
Subtype objects may be used wherever objects of a supertype are expected. This is known as substitution.

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:

```
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:

```
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?

```
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:

```
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?

```
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**:

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

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

```
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

```
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:

```
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*:

```
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.